synthcity

vanderschaar-lab

Mar 11, 2024

CONTENTS:

CHAPTER

ONE

FEATURES:

- Easy to extend pluginable architecture.
- Several evaluation metrics for correctness and privacy.
- Several reference models, by type:
  - General purpose: GAN-based (AdsGAN, CTGAN, PATEGAN, DP-GAN),VAE-based(TVAE, RTVAE), Normalizing flows, Bayesian Networks(PrivBayes, BN).
  - Time Series & Time-Series Survival generators: TimeGAN, FourierFlows, TimeVAE.
  - Static Survival Analysis: SurvivalGAN, SurVAE.
  - Privacy-focused: DECAF, DP-GAN, AdsGAN, PATEGAN, PrivBayes.
  - Domain adaptation: RadialGAN.
  - Images: Image ConditionalGAN, Image AdsGAN.
- Read the docs !
- Checkout the tutorials!

CHAPTER

TWO

INSTALLATION

The library can be installed from PyPI using

```
$ pip install synthcity
```

or from source, using

```
$ pip install .
```

CHAPTER

THREE

SAMPLE USAGE

3.1 Generic data

- List the available general-purpose generators

```
from synthcity.plugins import Plugins

Plugins(categories=["generic", "privacy"]).list()
```

- Load and train a tabular generator

```
from sklearn.datasets import load_diabetes
from synthcity.plugins import Plugins

X, y = load_diabetes(return_X_y=True, as_frame=True)
X["target"] = y

syn_model = Plugins().get("adsgan")

syn_model.fit(X)
```

- Generate new synthetic tabular data

```
syn_model.generate(count = 10)
```

- Benchmark the quality of the plugins

```
# third party
from sklearn.datasets import load_diabetes

# synthcity absolute
from synthcity.benchmark import Benchmarks
from synthcity.plugins.core.constraints import Constraints
from synthcity.plugins.core.dataloader import GenericDataLoader

X, y = load_diabetes(return_X_y=True, as_frame=True)
X["target"] = y

loader = GenericDataLoader(X, target_column="target", sensitive_columns=["sex"])

score = Benchmarks.evaluate(
```

(continues on next page)

(continued from previous page)

```
    [
        (f"example_{model}", model, {})  # testname, plugin name, plugin args
        for model in ["adsgan", "ctgan", "tvae"]
    ],
    loader,
    synthetic_size=1000,
    metrics={"performance": ["linear_model"]},
    repeats=3,
)
Benchmarks.print(score)
```

3.2 Static Survival analysis

- List the available generators dedicated to survival analysis

```
from synthcity.plugins import Plugins

Plugins(categories=["generic", "privacy", "survival_analysis"]).list()
```

- Generate new data

```
from lifelines.datasets import load_rossi
from synthcity.plugins.core.dataloader import SurvivalAnalysisDataLoader
from synthcity.plugins import Plugins

X = load_rossi()
data = SurvivalAnalysisDataLoader(
    X,
    target_column="arrest",
    time_to_event_column="week",
)

syn_model = Plugins().get("survival_gan")

syn_model.fit(data)

syn_model.generate(count=10)
```

3.3 Time series

- List the available generators

```
from synthcity.plugins import Plugins

Plugins(categories=["generic", "privacy", "time_series"]).list()
```

- Generate new data

```
# synthcity absolute
from synthcity.plugins import Plugins
from synthcity.plugins.core.dataloader import TimeSeriesDataLoader
from synthcity.utils.datasets.time_series.google_stocks import GoogleStocksDataloader

static_data, temporal_data, horizons, outcome = GoogleStocksDataloader().load()
data = TimeSeriesDataLoader(
    temporal_data=temporal_data,
    observation_times=horizons,
    static_data=static_data,
    outcome=outcome,
)

syn_model = Plugins().get("timegan")

syn_model.fit(data)

syn_model.generate(count=10)
```

3.4 Images

Note : The architectures used for generators are not state-of-the-art. For other architectures, consider extending the `suggest_image_generator_discriminator_arch` method from the `convnet.py` module.

- List the available generators

```
from synthcity.plugins import Plugins

Plugins(categories=["images"]).list()
```

- Generate new data ```python from synthcity.plugins import Plugins from synthcity.plugins.core.dataloader import ImageDataLoader from torchvision import datasets

dataset = datasets.MNIST(“.”, download=True) loader = ImageDataLoader(dataset).sample(100)

syn_model = Plugins().get(“image_cgan”)

syn_model.fit(loader)

syn_img, syn_labels = syn_model.generate(count=10).unpack().numpy()

print(syn_img.shape)

```
### Serialization

* Using save/load methods

```python
from synthcity.utils.serialization import save, load
from synthcity.plugins import Plugins

syn_model = Plugins().get("adsgan")

buff = save(syn_model)
```

(continues on next page)
```

(continued from previous page)

```
reloaded = load(buff)

assert syn_model.name() == reloaded.name()
```

- Saving and loading models from disk

```
from synthcity.utils.serialization import save_to_file, load_from_file
from synthcity.plugins import Plugins

syn_model = Plugins().get("adsgan")

save_to_file('./adsgan_10_epochs.pkl', syn_model)
reloaded = load_from_file('./adsgan_10_epochs.pkl')

assert syn_model.name() == reloaded.name()
```

- Using the Serializable interface

```
from synthcity.plugins import Plugins

syn_model = Plugins().get("adsgan")

buff = syn_model.save()
reloaded = Plugins().load(buff)

assert syn_model.name() == reloaded.name()
```

CHAPTER

FOUR

METHODS

4.1 Bayesian methods

Method	Description	Reference
bayesian_network	The method represents a set of random variables and their conditional dependencies via a directed acyclic graph (DAG), and uses it to sample new data points	pgmpy
privbayes	A differentially private method for releasing high-dimensional data.	PrivBayes: Private Data Release via Bayesian Networks

4.2 Generative adversarial networks(GANs)

Method	Description	Reference
ads-gan	A conditional GAN framework that generates synthetic data while minimize patient identifiability that is defined based on the probability of re-identification given the combination of all data on any individual patient	Anonymization Through Data Synthesis Using Generative Adversarial Networks (ADS-GAN)
pate-gan	The methos uses the Private Aggregation of Teacher Ensembles (PATE) framework and applies it to GANs, allowing to tightly bound the influence of any individual sample on the model, resulting in tight differential privacy guarantees and thus an improved performance over models with the same guarantees.	PATE-GAN: Generating Synthetic Data with Differential Privacy Guarantees
ct-gan	A conditional generative adversarial network which can handle tabular data.	Modeling Tabular data using Conditional GAN

4.3 Variational autoencoders(VAE)

Method	Description	Reference
tvae	A conditional VAE network which can handle tabular data.	Modeling Tabular data using Conditional GAN
rt-vae	A robust variational autoencoder with divergence for tabular data (RTVAE) with mixed categorical and continuous features.	Robust Variational Autoencoder for Tabular Data with Divergence

4.4 Normalizing Flows

Method	Description	Reference
nflow	Normalizing Flows are generative models which produce tractable distributions where both sampling and density evaluation can be efficient and exact.	Neural Spline Flows

4.5 Static Survival analysis methods

Method	Description	Reference
survival_gan	SurvivalGAN is a generative model that can handle survival data by addressing the imbalance in the censoring and time horizons, using a dedicated mechanism for approximating time to event/censoring from the input and survival function.	—
survival_ctgan	SurvivalGAN version using CTGAN	—
survae	SurvivalGAN version using VAE	—
survival_nflow	SurvivalGAN version using normalizing flows	—

4.6 Time-Series and Time-Series Survival Analysis methods

Method	Description	Reference
timegan	TimeGAN is a framework for generating realistic time-series data that combines the flexibility of the unsupervised paradigm with the control afforded by supervised training. Through a learned embedding space jointly optimized with both supervised and adversarial objectives, the network adheres to the dynamics of the training data during sampling.	Time-series Generative Adversarial Networks
fflows	FFlows is an explicit likelihood model based on a novel class of normalizing flows that view time-series data in the frequency-domain rather than the time-domain. The method uses a discrete Fourier transform (DFT) to convert variable-length time-series with arbitrary sampling periods into fixed-length spectral representations, then applies a (data-dependent) spectral filter to the frequency-transformed time-series.	Generative Time-series Modeling with Fourier Flows
probabilistic_ar	Probabilistic AutoRegressive model allows learning multi-type, multivariate timeseries data and later on generate new synthetic data that has the same format and properties as the learned one.	PAR model

4.7 Privacy & Fairness

Method	Description	Reference
decaf	Machine learning models have been criticized for reflecting unfair biases in the training data. Instead of solving this by introducing fair learning algorithms directly, DEACF focuses on generating fair synthetic data, such that any downstream learner is fair. Generating fair synthetic data from unfair data - while remaining truthful to the underlying data-generating process (DGP) - is non-trivial. DECAF is a GAN-based fair synthetic data generator for tabular data. With DECAF, we embed the DGP explicitly as a structural causal model in the input layers of the generator, allowing each variable to be reconstructed conditioned on its causal parents. This procedure enables inference time debiasing, where biased edges can be strategically removed to satisfy user-defined fairness requirements.	DECAF: Generating Fair Synthetic Data Using Causally-Aware Generative Networks
privbayes	A differentially private method for releasing high-dimensional data.	PrivBayes: Private Data Release via Bayesian Networks
dpgan	Differentially Private GAN	Differentially Private Generative Adversarial Network
adsgan	A conditional GAN framework that generates synthetic data while minimize patient identifiability that is defined based on the probability of re-identification given the combination of all data on any individual patient	Anonymization Through Data Synthesis Using Generative Adversarial Networks (ADS-GAN)
pategan	The methos uses the Private Aggregation of Teacher Ensembles (PATE) framework and applies it to GANs, allowing to tightly bound the influence of any individual sample on the model, resulting in tight differential privacy guarantees and thus an improved performance over models with the same guarantees.	PATE-GAN: Generating Synthetic Data with Differential Privacy Guarantees

4.8 Domain adaptation

Method	Description	Reference
radialgan	Training complex machine learning models for prediction often requires a large amount of data that is not always readily available. Leveraging these external datasets from related but different sources is, therefore, an essential task if good predictive models are to be built for deployment in settings where data can be rare. RadialGAN is an approach to the problem in which multiple GAN architectures are used to learn to translate from one dataset to another, thereby allowing to augment the target dataset effectively and learning better predictive models than just the target dataset.	RadialGAN: Leveraging multiple datasets to improve target-specific predictive models using Generative Adversarial Networks

4.9 Images

Method	Description	Reference
image_cgan	Conditional GAN for generating images	—
image_adsgan	The AdsGAN method adapted for image generation	—

4.10 Debug methods

Method	Description	Reference
marginal_distributions	A differentially private method that samples from the marginal distributions of the training set	—
uniform_sampler	A differentially private method that uniformly samples from the [min, max] ranges of each column.	—
dummy_sampler	Resample data points from the training set	—

CHAPTER

FIVE

EVALUATION METRICS

The following table contains the available evaluation metrics:

- **Sanity checks**

Metric	Description	Values
data_mismatch	Average number of columns with datatype(object, real, int) mismatch between the real and synthetic data	0: no datatype mismatch. 1: complete data type mismatch between the datasets.
common_rows_proportion	The proportion of rows in the real dataset leaked in the synthetic dataset.	0: there are no common rows between the real and synthetic datasets. 1: all the rows in the real dataset are leaked in the synthetic dataset.
nearest_syn_neighbor_distance	Average distance from the real data to the closest neighbor in the synthetic data	0: all the real rows are leaked in the synthetic dataset. 1: all the synthetic rows are far away from the real dataset.
close_values_probability	The probability of close values between the real and synthetic data.	0: there is no chance to have synthetic rows similar to the real.1 means that all the synthetic rows are similar to some real rows.
distant_values_probability	Average distance from the real data to the closest neighbor in the synthetic data	0: no chance to have rows in the synthetic far away from the real data. 1: all the synthetic datapoints are far away from the real data.

- **Statistical tests**

Metric	Description	Values
inverse_kl_divergence	The average inverse of the Kullback–Leibler Divergence	0: the datasets are from different distributions. 1: the datasets are from the same distribution.
ks_test	The Kolmogorov-Smirnov test	0: the distributions are totally different. 1: the distributions are identical.
chi_squared_test	The p-value. A small value indicates that we can reject the null hypothesis and that the distributions are different.	0: the distributions are different1: the distributions are identical.
max_mean_discrepancy	Empirical maximum mean discrepancy.	0: The distributions are the same. 1: The distributions are totally different.
jensen_shannon_dist	The Jensen-Shannon distance (metric) between two probability arrays. This is the square root of the Jensen-Shannon divergence.	0: The distributions are the same. 1: The distributions are totally different.
wasserstein_dist	Wasserstein Distance is a measure of the distance between two probability distributions.	0: The distributions are the same.
prdc	Computes precision, recall, density, and coverage given two manifolds.	—
alpha_precision	Evaluate the alpha-precision, beta-recall, and authenticity scores.	—
survival_km_distance	The distance between two Kaplan-Meier plots(survival analysis).	—
fid	The Frechet Inception Distance (FID) calculates the distance between two distributions of images.	—

- **Synthetic Data quality**

Metric	Description	Values
performance.xgb	Train an XGBoost classifier/regressor/survival model on real data(gt) and synthetic data(syn), and evaluate the performance on the test set.	1 for ideal performance, 0 for worst performance
performance.linear	Train a Linear classifier/regressor/survival model on real data(gt) and the synthetic data and evaluate the performance on test data.	1 for ideal performance, 0 for worst performance
performance.mlp	Train a Neural Net classifier/regressor/survival model on the real data and the synthetic data and evaluate the performance on test data.	1 for ideal performance, 0 for worst performance
performance.feat_rank_distance	Train a model on the synthetic data and a model on the real data. Compute the feature importance of the models on the same test data, and compute the rank distance between the importance(kendalltau or spearman)	1: similar ranks in the feature importance. 0: uncorrelated feature importance
detection_gmm	Train a GaussianMixture model to differentiate the synthetic data from the real data.	0: The datasets are indistinguishable. 1: The datasets are totally distinguishable.
detection_xgb	Train an XGBoost model to differentiate the synthetic data from the real data.	0: The datasets are indistinguishable. 1: The datasets are totally distinguishable.
detection_mlp	Train a Neural net to differentiate the synthetic data from the real data.	0: The datasets are indistinguishable. 1: The datasets are totally distinguishable.
detection_linear	Train a Linear model to differentiate the synthetic data from the real data.	0: The datasets are indistinguishable. 1: The datasets are totally distinguishable.

- **Privacy metrics**

Quasi-identifiers : pieces of information that are not of themselves unique identifiers, but are sufficiently well correlated with an entity that they can be combined with other quasi-identifiers to create a unique identifier.

Metric	Description	Values
k_anonymization	The minimum value k which satisfies the k-anonymity rule: each record is similar to at least another k-1 other records on the potentially identifying variables.	Reported on both the real and synthetic data.
l_diversity	The minimum value l which satisfies the l-diversity rule: every generalized block has to contain at least l different sensitive values.	Reported on both the real and synthetic data.
kmap	The minimum value k which satisfies the k-map rule: every combination of values for the quasi-identifiers appears at least k times in the reidentification(synthetic) dataset.	Reported on both the real and synthetic data.
delta_presence	The maximum re-identification risk for the real dataset from the synthetic dataset.	0 for no risk.
identifiability_score	The re-identification score on the real dataset from the synthetic dataset.	—]
sensitive_data_reidentification_xgb	Sensitive data prediction from the quasi-identifiers using an XGBoost.	0 for no risk.
sensitive_data_reidentification_mlp	Sensitive data prediction from the quasi-identifiers using a Neural Net.	0 for no risk.

CHAPTER

SIX

TESTS

Install the testing dependencies using

```
pip install .[testing]
```

The tests can be executed using

```
pytest -vsx
```

CHAPTER

SEVEN

CITING

If you use this code, please cite the associated paper:

```
@misc{https://doi.org/10.48550/arxiv.2301.07573,
  doi = {10.48550/ARXIV.2301.07573},
  url = {https://arxiv.org/abs/2301.07573},
  author = {Qian, Zhaozhi and Cebere, Bogdan-Constantin and van der Schaar, Mihaela},
  keywords = {Machine Learning (cs.LG), Artificial Intelligence (cs.AI), FOS: Computer␣
↪and information sciences, FOS: Computer and information sciences},
  title = {Synthcity: facilitating innovative use cases of synthetic data in different␣
↪data modalities},
  year = {2023},
  copyright = {Creative Commons Attribution 4.0 International}
}
```

CHAPTER

EIGHT

EXAMPLES

8.1 Tutorials

8.1.1 Getting started

8.1.2 General-purpose generators

synthcity.plugins.generic.plugin_goggle module

Reference: "GOGGLE: Generative Modelling for Tabular Data by Learning Relational Structure" Authors: Tennison Liu, Zhaozhi Qian, Jeroen Berrevoets, Mihaela van der Schaar

class GOGGLEPlugin(*n_iter: int = 1000*, *encoder_dim: int = 64*, *encoder_l: int = 2*, *het_encoding: bool = True*, *encoder_nonlin: str = 'tanh'*, *decoder_nonlin: str = 'tanh'*, *decoder_dim: int = 64*, *decoder_l: int = 2*, *data_encoder_max_clusters: int = 10*, *threshold: float = 0.1*, *decoder_arch: str = 'gcn'*, *graph_prior: Optional[numpy.ndarray] = None*, *prior_mask: Optional[numpy.ndarray] = None*, *alpha: float = 0.1*, *beta: float = 0.1*, *iter_opt: bool = True*, *learning_rate: float = 0.005*, *weight_decay: float = 0.001*, *batch_size: int = 32*, *patience: int = 50*, *logging_epoch: int = 100*, *device: Union[str, torch.device] = device(type='cpu')*, *random_state: int = 0*, *sampling_patience: int = 500*, *workspace: pathlib.Path = PosixPath('workspace')*, *compress_dataset: bool = False*, *dataloader_sampler: Optional[torch.utils.data.sampler.Sampler] = None*, ***kwargs: Any*)

Bases: *synthcity.plugins.core.plugin.Plugin*

Args:

Example

```
>>> from sklearn.datasets import load_iris
>>> from synthcity.plugins import Plugins
>>>
>>> X, y = load_iris(as_frame = True, return_X_y = True)
>>> X["target"] = y
>>>
>>> plugin = Plugins().get("goggle", n_iter = 100)
>>> plugin.fit(X)
>>>
>>> plugin.generate(50)
```

class Config
Bases: object

arbitrary_types_allowed = True

validate_assignment = True

fit(*X: Union[synthcity.plugins.core.dataloader.DataLoader, pandas.core.frame.DataFrame]*, **args: Any*, ***kwargs: Any*) → Any
Training method the synthetic data plugin.

Parameters

- **X** – DataLoader. The reference dataset.
- **cond** – Optional, Union[pd.DataFrame, pd.Series, np.ndarray] Optional Training Conditional. The training conditional can be used to control to output of some models, like GANs or VAEs. The content can be anything, as long as it maps to the training dataset X. Usage example:

```
>>> from sklearn.datasets import load_iris
>>> from synthcity.plugins.core.dataloader import GenericDataLoader
>>> from synthcity.plugins.core.constraints import Constraints
>>>
>>> # Load in `test_plugin` the generative model of choice
>>> # ....
>>>
>>> X, y = load_iris(as_frame=True, return_X_y=True)
>>> X["target"] = y
>>>
>>> X = GenericDataLoader(X)
>>> test_plugin.fit(X, cond=y)
>>>
>>> count = 10
>>> X_gen = test_plugin.generate(count, cond=np.ones(count))
>>>
>>> # The Conditional only optimizes the output generation
>>> # for GANs and VAEs, but does NOT guarantee the samples
>>> # are only from that condition.
>>> # If you want to guarantee that output contains only
>>> # "target" == 1 samples, use Constraints.
>>>
>>> constraints = Constraints(
>>>     rules=[
>>>         ("target", "==", 1),
>>>     ]
>>> )
>>> X_gen = test_plugin.generate(count,
>>>         cond=np.ones(count),
>>>         constraints=constraints
>>>        )
>>> assert (X_gen["target"] == 1).all()
```

Returns self

classmethod fqdn() → str
The Fully-Qualified name of the plugin.

generate(*count: Optional[int] = None*, *constraints: Optional[synthcity.plugins.core.constraints.Constraints] = None*, *random_state: Optional[int] = None*, ***kwargs: Any*) → *synthcity.plugins.core.dataloader.DataLoader*

Synthetic data generation method.

Parameters

- **count** – optional int. The number of samples to generate. If None, it generated len(reference_dataset) samples.
- **cond** – Optional, Union[pd.DataFrame, pd.Series, np.ndarray]. Optional Generation Conditional. The conditional can be used only if the model was trained using a conditional too. If provided, it must have *count* length. Not all models support conditionals. The conditionals can be used in VAEs or GANs to speed-up the generation under some constraints. For model agnostic solutions, check out the *constraints* parameter.
- **constraints** – optional Constraints. Optional constraints to apply on the generated data. If none, the reference schema constraints are applied. The constraints are model agnostic, and will filter the output of the generative model. The constraints are a list of rules. Each rule is a tuple of the form (<feature>, <operation>, <value>).

  Valid Operations:

  - "<", "lt" : less than <value>
  - "<=", "le": less or equal with <value>
  - ">", "gt" : greater than <value>
  - ">=", "ge": greater or equal with <value>
  - "==", "eq": equal with <value>
  - "in": valid for categorical features, and <value> must be array. for example, ("target", "in", [0, 1])
  - "dtype": <value> can be a data type. For example, ("target", "dtype", "int")

  Usage example:

```
>>> from synthcity.plugins.core.constraints import Constraints
>>> constraints = Constraints(
>>>   rules=[
>>>             ("InterestingFeature", "==", 0),
>>>         ]
>>>     )
>>>
>>> syn_data = syn_model.generate(
        count=count,
        constraints=constraints
    ).dataframe()
>>>
>>> assert (syn_data["InterestingFeature"] == 0).all()
```

- **random_state** – optional int. Optional random seed to use.

Returns <count> synthetic samples

static hyperparameter_space(***kwargs: Any*) → List[*synthcity.plugins.core.distribution.Distribution*]

Returns the hyperparameter space for the derived plugin.

static load(*buff: bytes*) → Any

static load_dict(*representation: dict*) → Any

static name() → str
The name of the plugin.

plot(*plt: Any*, *X:* synthcity.plugins.core.dataloader.DataLoader, *count: Optional[int] = None*, *plots: list = ['marginal', 'associations', 'tsne']*, ***kwargs: Any*) → Any
Plot the real-synthetic distributions.

Parameters

- **plt** – output
- **X** – DataLoader. The reference dataset.

Returns self

classmethod sample_hyperparameters(**args: Any*, ***kwargs: Any*) → Dict[str, Any]
Sample value from the hyperparameter space for the current plugin.

classmethod sample_hyperparameters_optuna(*trial: Any*, **args: Any*, ***kwargs: Any*) → Dict[str, Any]

save() → bytes

save_dict() → dict

save_to_file(*path: pathlib.Path*) → bytes

schema() → *synthcity.plugins.core.schema.Schema*
The reference schema

schema_includes(*other: Union[*synthcity.plugins.core.dataloader.DataLoader*, pandas.core.frame.DataFrame]*) → bool
Helper method to test if the reference schema includes a Dataset

Parameters **other** – DataLoader. The dataset to test

Returns bool, if the schema includes the dataset or not.

training_schema() → *synthcity.plugins.core.schema.Schema*
The internal schema

static type() → str
The type of the plugin.

static version() → str
API version

plugin
alias of *synthcity.plugins.generic.plugin_goggle.GOGGLEPlugin*

synthcity.plugins.generic.plugin_arf module

Reference: "Adversarial random forests for density estimation and generative modeling" Authors: David S. Watson, Kristin Blesch, Jan Kapar, and Marvin N. Wright

class ARFPlugin(*num_trees: int = 30*, *delta: int = 0*, *max_iters: int = 10*, *early_stop: bool = True*, *verbose: bool = True*, *min_node_size: int = 5*, *device: Union[str, torch.device] = device(type='cpu')*, *random_state: int = 0*, *sampling_patience: int = 500*, *workspace: pathlib.Path = PosixPath('workspace')*, *compress_dataset: bool = False*, ***kwargs: Any*)
Bases: *synthcity.plugins.core.plugin.Plugin*

Args:

Example

```
>>> from sklearn.datasets import load_iris
>>> from synthcity.plugins import Plugins
>>>
>>> X, y = load_iris(as_frame = True, return_X_y = True)
>>> X["target"] = y
>>>
>>> plugin = Plugins().get("arf")
>>> plugin.fit(X)
>>>
>>> plugin.generate(50)
```

class Config

Bases: object

arbitrary_types_allowed = True

validate_assignment = True

fit(*X: Union[*synthcity.plugins.core.dataloader.DataLoader*, pandas.core.frame.DataFrame]*, **args: Any*, ***kwargs: Any*) → Any

Training method the synthetic data plugin.

Parameters

- **X** – DataLoader. The reference dataset.
- **cond** – Optional, Union[pd.DataFrame, pd.Series, np.ndarray] Optional Training Conditional. The training conditional can be used to control to output of some models, like GANs or VAEs. The content can be anything, as long as it maps to the training dataset X. Usage example:

```
>>> from sklearn.datasets import load_iris
>>> from synthcity.plugins.core.dataloader import␣
↪GenericDataLoader
>>> from synthcity.plugins.core.constraints import Constraints
>>>
>>> # Load in `test_plugin` the generative model of choice
>>> # ....
>>>
>>> X, y = load_iris(as_frame=True, return_X_y=True)
>>> X["target"] = y
>>>
```

(continues on next page)

(continued from previous page)

```
>>> X = GenericDataLoader(X)
>>> test_plugin.fit(X, cond=y)
>>>
>>> count = 10
>>> X_gen = test_plugin.generate(count, cond=np.ones(count))
>>>
>>> # The Conditional only optimizes the output generation
>>> # for GANs and VAEs, but does NOT guarantee the samples
>>> # are only from that condition.
>>> # If you want to guarantee that output contains only
>>> # "target" == 1 samples, use Constraints.
>>>
>>> constraints = Constraints(
>>>     rules=[
>>>         ("target", "==", 1),
>>>     ]
>>> )
>>> X_gen = test_plugin.generate(count,
>>>         cond=np.ones(count),
>>>         constraints=constraints
>>>        )
>>> assert (X_gen["target"] == 1).all()
```

Returns self

classmethod fqdn() → str
The Fully-Qualified name of the plugin.

generate(*count: Optional[int] = None*, *constraints: Optional[*synthcity.plugins.core.constraints.Constraints*] = None*, *random_state: Optional[int] = None*, ***kwargs: Any*) → *synthcity.plugins.core.dataloader.DataLoader*
Synthetic data generation method.

Parameters

- **count** – optional int. The number of samples to generate. If None, it generated len(reference_dataset) samples.
- **cond** – Optional, Union[pd.DataFrame, pd.Series, np.ndarray]. Optional Generation Conditional. The conditional can be used only if the model was trained using a conditional too. If provided, it must have *count* length. Not all models support conditionals. The conditionals can be used in VAEs or GANs to speed-up the generation under some constraints. For model agnostic solutions, check out the *constraints* parameter.
- **constraints** – optional Constraints. Optional constraints to apply on the generated data. If none, the reference schema constraints are applied. The constraints are model agnostic, and will filter the output of the generative model. The constraints are a list of rules. Each rule is a tuple of the form (<feature>, <operation>, <value>).

  Valid Operations:

  - "<", "lt" : less than <value>
  - "<=", "le": less or equal with <value>
  - ">", "gt" : greater than <value>
  - ">=", "ge": greater or equal with <value>

- "==", "eq": equal with <value>
- "in": valid for categorical features, and <value> must be array. for example, ("target", "in", [0, 1])
- "dtype": <value> can be a data type. For example, ("target", "dtype", "int")

Usage example:

```
>>> from synthcity.plugins.core.constraints import Constraints
>>> constraints = Constraints(
>>>   rules=[
>>>             ("InterestingFeature", "==", 0),
>>>         ]
>>>     )
>>>
>>> syn_data = syn_model.generate(
        count=count,
        constraints=constraints
    ).dataframe()
>>>
>>> assert (syn_data["InterestingFeature"] == 0).all()
```

- **random_state** – optional int. Optional random seed to use.

Returns <count> synthetic samples

static hyperparameter_space(***kwargs: Any*) → List[*synthcity.plugins.core.distribution.Distribution*]
Returns the hyperparameter space for the derived plugin.

static load(*buff: bytes*) → Any

static load_dict(*representation: dict*) → Any

static name() → str
The name of the plugin.

plot(*plt: Any*, *X:* synthcity.plugins.core.dataloader.DataLoader, *count: Optional[int] = None*, *plots: list = ['marginal', 'associations', 'tsne']*, ***kwargs: Any*) → Any
Plot the real-synthetic distributions.

Parameters

- **plt** – output
- **X** – DataLoader. The reference dataset.

Returns self

classmethod sample_hyperparameters(**args: Any*, ***kwargs: Any*) → Dict[str, Any]
Sample value from the hyperparameter space for the current plugin.

classmethod sample_hyperparameters_optuna(*trial: Any*, **args: Any*, ***kwargs: Any*) → Dict[str, Any]

save() → bytes

save_dict() → dict

save_to_file(*path: pathlib.Path*) → bytes

schema() → *synthcity.plugins.core.schema.Schema*
The reference schema

schema_includes(*other: Union[*synthcity.plugins.core.dataloader.DataLoader, *pandas.core.frame.DataFrame]*) → bool
Helper method to test if the reference schema includes a Dataset

Parameters **other** – DataLoader. The dataset to test

Returns bool, if the schema includes the dataset or not.

training_schema() → *synthcity.plugins.core.schema.Schema*
The internal schema

static type() → str
The type of the plugin.

static version() → str
API version

plugin
alias of *synthcity.plugins.generic.plugin_arf.ARFPlugin*

synthcity.plugins.generic.plugin_great module

8.1.3 Time-series generators

8.1.4 Privacy-related generators

8.1.5 Domain adaptation generators

8.1.6 Images

CHAPTER

NINE

DATALOADERS AND DATASETS

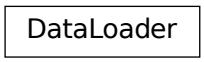
9.1 Datasets and DataLoaders

9.1.1 Dataloaders and Datasets

synthcity.plugins.core.dataloader module

class DataLoader(*data_type: str*, *data: Any*, *static_features: List[str] = []*, *temporal_features: List[str] = []*, *sensitive_features: List[str] = []*, *important_features: List[str] = []*, *outcome_features: List[str] = []*, *train_size: float = 0.8*, *random_state: int = 0*, ***kwargs: Any*)

Bases: object



Base class for all data loaders.

Each derived class must implement the following methods: unpack() - a method that unpacks the columns and returns features and labels (X, y). decorate() - a method that creates a new instance of DataLoader by decorating the input data with the same DataLoader properties (e.g. sensitive features, target column, etc.) dataframe() - a method that returns the pandas dataframe that contains all features and samples numpy() - a method that returns the numpy array that contains all features and samples info() - a method that returns a dictionary of DataLoader information __len__() - a method that returns the number of samples in the DataLoader satisfies() - a method that tests if the current DataLoader satisfies the constraint provided match() - a method that returns a new DataLoader where the provided constraints are met from_info() - a static method that creates a DataLoader from the data and the information dictionary sample() - returns a new DataLoader that contains a random subset of N samples drop() - returns a new DataLoader with a list of columns dropped __getitem__() - getting features by names __setitem__() - setting features by names train() - returns a DataLoader containing the training set test() - returns a DataLoader containing the testing set fillna() - returns a DataLoader with NaN filled by the provided number(s)

If any method implementation is missing, the class constructor will fail.

Constructor Args:

data_type: str The type of DataLoader, currently supports “generic”, “time_series” and “survival”.

data: Any The object that contains the data

static_features: List[str] List of feature names that are static features (as opposed to temporal features).

temporal_features: List of feature names that are temporal features, i.e. observed over time.

sensitive_features: List[str] Name of sensitive features.

important_features: List[str] Default: None. Only relevant for SurvivalGAN method.

outcome_features: The feature name that provides labels for downstream tasks.

abstract property columns: list

compress() → Tuple[*synthcity.plugins.core.dataloader.DataLoader*, Dict]

abstract compression_protected_features() → list

abstract dataframe() → pandas.core.frame.DataFrame

decode(*encoders: Dict[str, Any]*) → *synthcity.plugins.core.dataloader.DataLoader*

decompress(*context: Dict*) → *synthcity.plugins.core.dataloader.DataLoader*

abstract decorate(*data: Any*) → *synthcity.plugins.core.dataloader.DataLoader*

domain() → Optional[str]

abstract drop(*columns: list = []*) → *synthcity.plugins.core.dataloader.DataLoader*

encode(*encoders: Optional[Dict[str, Any]] = None*) → Tuple[*synthcity.plugins.core.dataloader.DataLoader*, Dict]

abstract fillna(*value: Any*) → *synthcity.plugins.core.dataloader.DataLoader*

abstract static from_info(*data: pandas.core.frame.DataFrame*, *info: dict*) → *synthcity.plugins.core.dataloader.DataLoader*

abstract get_fairness_column() → Union[str, Any]

hash() → str

abstract info() → dict

abstract is_tabular() → bool

abstract match(*constraints:* synthcity.plugins.core.constraints.Constraints) → *synthcity.plugins.core.dataloader.DataLoader*

abstract numpy() → numpy.ndarray

raw() → Any

abstract sample(*count: int*, *random_state: int = 0*) → *synthcity.plugins.core.dataloader.DataLoader*

abstract satisfies(*constraints:* synthcity.plugins.core.constraints.Constraints) → bool

abstract property shape: tuple

abstract test() → *synthcity.plugins.core.dataloader.DataLoader*

abstract train() → *synthcity.plugins.core.dataloader.DataLoader*

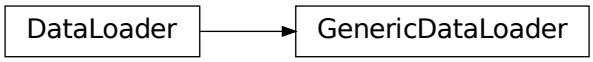
type() → str

abstract unpack(*as_numpy: bool = False*, *pad: bool = False*) → Any

property values: numpy.ndarray

class GenericDataLoader(*data: Union[pandas.core.frame.DataFrame, list, numpy.ndarray]*, *sensitive_features: List[str] = []*, *important_features: List[str] = []*, *target_column: Optional[str] = None*, *fairness_column: Optional[str] = None*, *domain_column: Optional[str] = None*, *random_state: int = 0*, *train_size: float = 0.8*, ***kwargs: Any*)

Bases: *synthcity.plugins.core.dataloader.DataLoader*



Data loader for generic tabular data.

Constructor Args:

data: Union[pd.DataFrame, list, np.ndarray] The dataset. Either a Pandas DataFrame or a Numpy Array.

sensitive_features: List[str] Name of sensitive features.

important_features: List[str] Default: None. Only relevant for SurvivalGAN method.

target_column: Optional[str] The feature name that provides labels for downstream tasks.

fairness_column: Optional[str] Optional fairness column label, used for fairness benchmarking.

domain_column: Optional[str] Optional domain label, used for domain adaptation algorithms.

random_state: int Defaults to zero.

Example

```
>>> from sklearn.datasets import load_diabetes
>>> from synthcity.plugins.core.dataloader import GenericDataLoader
>>> X, y = load_diabetes(return_X_y=True, as_frame=True)
>>> X["target"] = y
>>> # Important note: preprocessing data with OneHotEncoder or StandardScaler is␣
↪not needed or recommended.
>>> # Synthcity handles feature encoding and standardization internally.
>>> loader = GenericDataLoader(X, target_column="target", sensitive_columns=["sex"],
↪)
```

property columns: list

compress() → Tuple[*synthcity.plugins.core.dataloader.DataLoader*, Dict]

compression_protected_features() → list

dataframe() → pandas.core.frame.DataFrame

decode(*encoders: Dict[str, Any]*) → *synthcity.plugins.core.dataloader.DataLoader*

decompress(*context: Dict*) → *synthcity.plugins.core.dataloader.DataLoader*

decorate(*data: Any*) → *synthcity.plugins.core.dataloader.DataLoader*

domain() → Optional[str]

drop(*columns: list = []*) → *synthcity.plugins.core.dataloader.DataLoader*

encode(*encoders: Optional[Dict[str, Any]] = None*) → Tuple[*synthcity.plugins.core.dataloader.DataLoader*, Dict]

fillna(*value: Any*) → *synthcity.plugins.core.dataloader.DataLoader*

static from_info(*data: pandas.core.frame.DataFrame*, *info: dict*) → *synthcity.plugins.core.dataloader.GenericDataLoader*

get_fairness_column() → Union[str, Any]

hash() → str

info() → dict

is_tabular() → bool

match(*constraints:* synthcity.plugins.core.constraints.Constraints) → *synthcity.plugins.core.dataloader.DataLoader*

numpy() → numpy.ndarray

raw() → Any

sample(*count: int*, *random_state: int = 0*) → *synthcity.plugins.core.dataloader.DataLoader*

satisfies(*constraints:* synthcity.plugins.core.constraints.Constraints) → bool

property shape: tuple

test() → *synthcity.plugins.core.dataloader.DataLoader*

train() → *synthcity.plugins.core.dataloader.DataLoader*

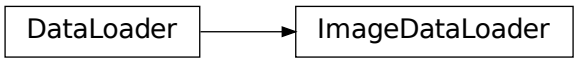
type() → str

unpack(*as_numpy: bool = False*, *pad: bool = False*) → Any

property values: numpy.ndarray

class ImageDataLoader(*data: Union[torch.utils.data.dataset.Dataset, Tuple[torch.Tensor, torch.Tensor]]*, *height: int = 32*, *width: Optional[int] = None*, *random_state: int = 0*, *train_size: float = 0.8*, ***kwargs: Any*)

Bases: *synthcity.plugins.core.dataloader.DataLoader*



Data loader for generic image data.

Constructor Args:

data: torch.utils.data.Dataset or torch.Tensor The image dataset or a tuple of (tensor images, tensor labels)

random_state: int Defaults to zero.

height: int. Default = 32 Height to use internally

width: Optional[int] Optional width to use internally. If None, it is used the same value as height.

train_size: float = 0.8 Train dataset ratio.

Example

```
>>> dataset = datasets.MNIST(".", download=True)
>>>
>>> loader = ImageDataLoader(
>>>     data=dataset,
>>>     train_size=0.8,
>>>     height=32,
>>>     width=w32,
>>> )
```

property columns: list

compress() → Tuple[*synthcity.plugins.core.dataloader.DataLoader*, Dict]

compression_protected_features() → list

dataframe() → pandas.core.frame.DataFrame

decode(*encoders: Dict[str, Any]*) → *synthcity.plugins.core.dataloader.DataLoader*

decompress(*context: Dict*) → *synthcity.plugins.core.dataloader.DataLoader*

decorate(*data: Any*) → *synthcity.plugins.core.dataloader.DataLoader*

domain() → Optional[str]

drop(*columns: list = []*) → *synthcity.plugins.core.dataloader.DataLoader*

encode(*encoders: Optional[Dict[str, Any]] = None*) → Tuple[*synthcity.plugins.core.dataloader.DataLoader*, Dict]

fillna(*value: Any*) → *synthcity.plugins.core.dataloader.DataLoader*

static from_info(*data: torch.utils.data.dataset.Dataset*, *info: dict*) → *synthcity.plugins.core.dataloader.ImageDataLoader*

get_fairness_column() → None
Not implemented for ImageDataLoader

hash() → str

info() → dict

is_tabular() → bool

match(*constraints:* synthcity.plugins.core.constraints.Constraints) → *synthcity.plugins.core.dataloader.DataLoader*

numpy() → numpy.ndarray

raw() → Any

sample(*count: int*, *random_state: int = 0*) → *synthcity.plugins.core.dataloader.DataLoader*

satisfies(*constraints:* synthcity.plugins.core.constraints.Constraints) → bool

property shape: tuple

test() → *synthcity.plugins.core.dataloader.DataLoader*

train() → *synthcity.plugins.core.dataloader.DataLoader*

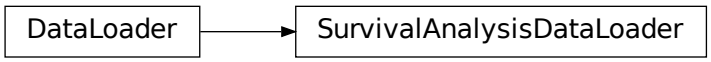
type() → str

unpack(*as_numpy: bool = False*, *pad: bool = False*) → Any

property values: numpy.ndarray

class SurvivalAnalysisDataLoader(*data: pandas.core.frame.DataFrame*, *time_to_event_column: str*, *target_column: str*, *time_horizons: list = []*, *sensitive_features: List[str] = []*, *important_features: List[str] = []*, *fairness_column: Optional[str] = None*, *random_state: int = 0*, *train_size: float = 0.8*, ***kwargs: Any*)

Bases: *synthcity.plugins.core.dataloader.DataLoader*



Data Loader for Survival Analysis Data

Constructor Args:

data: Union[pd.DataFrame, list, np.ndarray] The dataset. Either a Pandas DataFrame or a Numpy Array.

time_to_event_column: str Survival Analysis specific time-to-event feature

target_column: str The outcome: event or censoring.

sensitive_features: List[str] Name of sensitive features.

important_features: List[str] Default: None. Only relevant for SurvivalGAN method.

target_column: str The feature name that provides labels for downstream tasks.

fairness_column: Optional[str] Optional fairness column label, used for fairness benchmarking.

domain_column: Optional[str] Optional domain label, used for domain adaptation algorithms.

random_state: int Defaults to zero.

train_size: float The ratio to use for train splits.

Example

```
>>> TODO
```

property columns: list

compress() → Tuple[*synthcity.plugins.core.dataloader.DataLoader*, Dict]

compression_protected_features() → list

dataframe() → pandas.core.frame.DataFrame

decode(*encoders: Dict[str, Any]*) → *synthcity.plugins.core.dataloader.DataLoader*

decompress(*context: Dict*) → *synthcity.plugins.core.dataloader.DataLoader*

decorate(*data: Any*) → *synthcity.plugins.core.dataloader.DataLoader*

domain() → Optional[str]

drop(*columns: list = []*) → *synthcity.plugins.core.dataloader.DataLoader*

encode(*encoders: Optional[Dict[str, Any]] = None*) → Tuple[*synthcity.plugins.core.dataloader.DataLoader*, Dict]

fillna(*value: Any*) → *synthcity.plugins.core.dataloader.DataLoader*

static from_info(*data: pandas.core.frame.DataFrame*, *info: dict*) → *synthcity.plugins.core.dataloader.DataLoader*

get_fairness_column() → Union[str, Any]

hash() → str

info() → dict

is_tabular() → bool

match(*constraints:* synthcity.plugins.core.constraints.Constraints) → *synthcity.plugins.core.dataloader.DataLoader*

numpy() → numpy.ndarray

raw() → Any

sample(*count: int*, *random_state: int = 0*) → *synthcity.plugins.core.dataloader.DataLoader*

satisfies(*constraints:* synthcity.plugins.core.constraints.Constraints) → bool

property shape: tuple

test() → *synthcity.plugins.core.dataloader.DataLoader*

train() → *synthcity.plugins.core.dataloader.DataLoader*

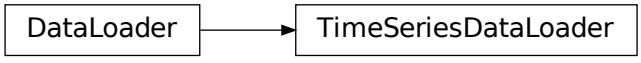
type() → str

unpack(*as_numpy: bool = False*, *pad: bool = False*) → Any

property values: numpy.ndarray

class TimeSeriesDataLoader(*temporal_data: List[pandas.core.frame.DataFrame]*, *observation_times: List*, *outcome: Optional[pandas.core.frame.DataFrame] = None*, *static_data: Optional[pandas.core.frame.DataFrame] = None*, *sensitive_features: List[str] = []*, *important_features: List[str] = []*, *fairness_column: Optional[str] = None*, *random_state: int = 0*, *train_size: float = 0.8*, *seq_offset: int = 0*, ***kwargs: Any*)

Bases: *synthcity.plugins.core.dataloader.DataLoader*



Data Loader for Time Series Data

Constructor Args:

temporal data: List[pd.DataFrame] The temporal data. A list of pandas DataFrames

observation times: List List of arrays mapping directly to index of each dataframe in temporal_data

outcome: Optional[pd.DataFrame] = None pandas DataFrame thatn can be anything (eg, labels, regression outcome)

static_data: Optional[pd.DataFrame] = None pandas DataFrame mapping directly to index of each dataframe in temporal_data

sensitive_features: List[str] Name of sensitive features

important_features List[str] Default: None. Only relevant for SurvivalGAN method

fairness_column: Optional[str] Optional fairness column label, used for fairness benchmarking.

random_state: int Defaults to zero.

Example

```
>>> TODO
```

property columns: list

compress() → Tuple[*synthcity.plugins.core.dataloader.DataLoader*, Dict]

compression_protected_features() → list

dataframe() → pandas.core.frame.DataFrame

decode(*encoders: Dict[str, Any]*) → *synthcity.plugins.core.dataloader.DataLoader*

decompress(*context: Dict*) → *synthcity.plugins.core.dataloader.DataLoader*

decorate(*data: Any*) → *synthcity.plugins.core.dataloader.DataLoader*

domain() → Optional[str]

drop(*columns: list = []*) → *synthcity.plugins.core.dataloader.DataLoader*

encode(*encoders: Optional[Dict[str, Any]] = None*) → Tuple[*synthcity.plugins.core.dataloader.DataLoader*, Dict]

static extract_masked_features(*full_temporal_features: list*) → tuple

fillna(*value: Any*) → *synthcity.plugins.core.dataloader.DataLoader*

filter_ids(*ids_list: list*) → pandas.core.frame.DataFrame

static from_info(*data: pandas.core.frame.DataFrame*, *info: dict*) → *synthcity.plugins.core.dataloader.DataLoader*

get_fairness_column() → Union[str, Any]

hash() → str

ids() → list

info() → dict

is_tabular() → bool

static mask_temporal_data(*temporal_data: List[pandas.core.frame.DataFrame]*, *observation_times: List*, *fill: Any = 0*) → Any

match(*constraints:* synthcity.plugins.core.constraints.Constraints) → *synthcity.plugins.core.dataloader.DataLoader*

numpy() → numpy.ndarray

static pack_raw_data(*static_data: Optional[pandas.core.frame.DataFrame]*, *temporal_data: List[pandas.core.frame.DataFrame]*, *observation_times: List*, *outcome: Optional[pandas.core.frame.DataFrame]*, *fill: Any = nan*, *seq_offset: int = 0*) → pandas.core.frame.DataFrame

static pad_and_mask(*static_data: Optional[pandas.core.frame.DataFrame]*, *temporal_data: List[pandas.core.frame.DataFrame]*, *observation_times: List*, *outcome: Optional[pandas.core.frame.DataFrame]*, *only_features: Any = False*, *fill: Any = 0*) → Any

static pad_raw_data(*static_data: Optional[pandas.core.frame.DataFrame]*, *temporal_data: List[pandas.core.frame.DataFrame]*, *observation_times: List*, *outcome: Optional[pandas.core.frame.DataFrame]*) → Any

static pad_raw_features(*static_data: Optional[pandas.core.frame.DataFrame]*, *temporal_data: List[pandas.core.frame.DataFrame]*, *observation_times: List*, *outcome: Optional[pandas.core.frame.DataFrame]*) → Any

raw() → Any

property raw_columns: list

sample(*count: int*, *random_state: int = 0*) → *synthcity.plugins.core.dataloader.DataLoader*

satisfies(*constraints:* synthcity.plugins.core.constraints.Constraints) → bool

static sequential_view(*static_data: Optional[pandas.core.frame.DataFrame]*, *temporal_data: List[pandas.core.frame.DataFrame]*, *observation_times: List*, *outcome: Optional[pandas.core.frame.DataFrame]*, *id_col: str = 'seq_id'*, *time_id_col: str = 'seq_time_id'*, *seq_offset: int = 0*) → Tuple[pandas.core.frame.DataFrame, dict]

property shape: tuple

test() → *synthcity.plugins.core.dataloader.DataLoader*

train() → *synthcity.plugins.core.dataloader.DataLoader*

type() → str

static unique_temporal_features(*temporal_data: List[pandas.core.frame.DataFrame]*) → List

static unmask_temporal_data(*temporal_data: List[pandas.core.frame.DataFrame]*, *observation_times: List*, *fill: Any = nan*) → Any

unpack(*as_numpy: bool = False*, *pad: bool = False*) → Any

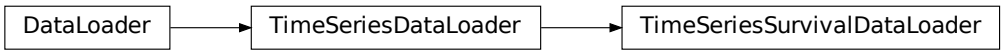
unpack_and_decorate(*data: pandas.core.frame.DataFrame*) → *synthcity.plugins.core.dataloader.DataLoader*

static unpack_raw_data(*data: pandas.core.frame.DataFrame*, *info: dict*) → Tuple[Optional[pandas.core.frame.DataFrame], List[pandas.core.frame.DataFrame], List, Optional[pandas.core.frame.DataFrame]]

property values: numpy.ndarray

class TimeSeriesSurvivalDataLoader(*temporal_data: List[pandas.core.frame.DataFrame], observation_times: Union[List, numpy.ndarray, pandas.core.series.Series], T: Union[pandas.core.series.Series, numpy.ndarray], E: Union[pandas.core.series.Series, numpy.ndarray], static_data: Optional[pandas.core.frame.DataFrame] = None, sensitive_features: List[str] = [], important_features: List[str] = [], time_horizons: list = [], fairness_column: Optional[str] = None, random_state: int = 0, train_size: float = 0.8, seq_offset: int = 0, **kwargs: Any*)

Bases: *synthcity.plugins.core.dataloader.TimeSeriesDataLoader*



Data loader for Time series survival data

Constructor Args:

temporal_data: List[pd.DataFrame} The temporal data. A list of pandas DataFrames.

observation_times: List List of arrays mapping directly to index of each dataframe in temporal_data

T: Union[pd.Series, np.ndarray, pd.Series] Time-to-event data

E: Union[pd.Series, np.ndarray, pd.Series] E is censored/event data

static_data Optional[pd.DataFrame] = None pandas DataFrame of static features for each subject

sensitive_features: List[str] Name of sensitive features

important_features: List[str} Default: None. Only relevant for SurvivalGAN method.

fairness_column: Optional[str] Optional fairness column label, used for fairness benchmarking.

random_state. int Defaults to zero.

Example

```
>>> TODO
```

property columns: list

compress() → Tuple[*synthcity.plugins.core.dataloader.DataLoader*, Dict]

compression_protected_features() → list

dataframe() → pandas.core.frame.DataFrame

decode(*encoders: Dict[str, Any]*) → *synthcity.plugins.core.dataloader.DataLoader*

decompress(*context: Dict*) → *synthcity.plugins.core.dataloader.DataLoader*

decorate(*data: Any*) → *synthcity.plugins.core.dataloader.DataLoader*

domain() → Optional[str]

drop(*columns: list = []*) → *synthcity.plugins.core.dataloader.DataLoader*

encode(*encoders: Optional[Dict[str, Any]] = None*) → Tuple[*synthcity.plugins.core.dataloader.DataLoader*, Dict]

static extract_masked_features(*full_temporal_features: list*) → tuple

fillna(*value: Any*) → *synthcity.plugins.core.dataloader.DataLoader*

filter_ids(*ids_list: list*) → pandas.core.frame.DataFrame

static from_info(*data: pandas.core.frame.DataFrame*, *info: dict*) → *synthcity.plugins.core.dataloader.DataLoader*

get_fairness_column() → Union[str, Any]

hash() → str

ids() → list

info() → dict

is_tabular() → bool

static mask_temporal_data(*temporal_data: List[pandas.core.frame.DataFrame]*, *observation_times: List*, *fill: Any = 0*) → Any

match(*constraints:* synthcity.plugins.core.constraints.Constraints) → *synthcity.plugins.core.dataloader.DataLoader*

numpy() → numpy.ndarray

static pack_raw_data(*static_data: Optional[pandas.core.frame.DataFrame]*, *temporal_data: List[pandas.core.frame.DataFrame]*, *observation_times: List*, *outcome: Optional[pandas.core.frame.DataFrame]*, *fill: Any = nan*, *seq_offset: int = 0*) → pandas.core.frame.DataFrame

static pad_and_mask(*static_data: Optional[pandas.core.frame.DataFrame]*, *temporal_data: List[pandas.core.frame.DataFrame]*, *observation_times: List*, *outcome: Optional[pandas.core.frame.DataFrame]*, *only_features: Any = False*, *fill: Any = 0*) → Any

static pad_raw_data(*static_data: Optional[pandas.core.frame.DataFrame]*, *temporal_data: List[pandas.core.frame.DataFrame]*, *observation_times: List*, *outcome: Optional[pandas.core.frame.DataFrame]*) → Any

static pad_raw_features(*static_data: Optional[pandas.core.frame.DataFrame]*, *temporal_data: List[pandas.core.frame.DataFrame]*, *observation_times: List*, *outcome: Optional[pandas.core.frame.DataFrame]*) → Any

raw() → Any

property raw_columns: list

sample(*count: int*, *random_state: int = 0*) → *synthcity.plugins.core.dataloader.DataLoader*

satisfies(*constraints:* synthcity.plugins.core.constraints.Constraints) → bool

static sequential_view(*static_data: Optional[pandas.core.frame.DataFrame]*, *temporal_data: List[pandas.core.frame.DataFrame]*, *observation_times: List*, *outcome: Optional[pandas.core.frame.DataFrame]*, *id_col: str = 'seq_id'*, *time_id_col: str = 'seq_time_id'*, *seq_offset: int = 0*) → Tuple[pandas.core.frame.DataFrame, dict]

property shape: tuple

test() → *synthcity.plugins.core.dataloader.DataLoader*

train() → *synthcity.plugins.core.dataloader.DataLoader*

type() → str

static unique_temporal_features(*temporal_data: List[pandas.core.frame.DataFrame]*) → List

static unmask_temporal_data(*temporal_data: List[pandas.core.frame.DataFrame]*, *observation_times: List*, *fill: Any = nan*) → Any

unpack(*as_numpy: bool = False*, *pad: bool = False*) → Any

unpack_and_decorate(*data: pandas.core.frame.DataFrame*) → *synthcity.plugins.core.dataloader.DataLoader*

static unpack_raw_data(*data: pandas.core.frame.DataFrame*, *info: dict*) → Tuple[Optional[pandas.core.frame.DataFrame], List[pandas.core.frame.DataFrame], List, Optional[pandas.core.frame.DataFrame]]

property values: numpy.ndarray

create_from_info(*data: Union[pandas.core.frame.DataFrame, torch.utils.data.dataset.Dataset]*, *info: dict*) → *synthcity.plugins.core.dataloader.DataLoader*

Helper for creating a DataLoader from existing information.

synthcity.plugins.core.dataset module

class ConditionalDataset(*data: torch.utils.data.dataset.Dataset*, *cond: Optional[torch.Tensor] = None*)

Bases: `torch.utils.data.dataset.Dataset`

Helper dataset for wrapping existing datasets with custom tensors

Parameters

- **data** – torch.Dataset
- **cond** – Optional Tensor

class FlexibleDataset(*data: torch.utils.data.dataset.Dataset*, *transform: Optional[torch.nn.modules.module.Module] = None*, *indices: Optional[list] = None*)

Bases: `torch.utils.data.dataset.Dataset`

Helper dataset wrapper for post-processing or transforming another dataset. Used for controlling the image sizes for the synthcity models.

The class supports adding custom transforms to existing datasets, and to subsample a set of indices.

Parameters

- **data** – torch.Dataset
- **transform** – An optional list of transforms
- **indices** – An optional list of indices to subsample

filter_indices(*indices: List[int]*) → *synthcity.plugins.core.dataset.FlexibleDataset*

labels() → numpy.ndarray

numpy() → Tuple[numpy.ndarray, numpy.ndarray]

shape() → Tuple

tensors() → Tuple[torch.Tensor, torch.Tensor]

class NumpyDataset(*X: numpy.ndarray*, *y: numpy.ndarray*)
Bases: torch.utils.data.dataset.Dataset

Helper class for wrapping Numpy arrays in torch Datasets :param X: np.ndarray :param y: np.ndarray

class TensorDataset(*images: torch.Tensor*, *targets: Optional[torch.Tensor]*)
Bases: torch.utils.data.dataset.Dataset

Helper dataset for wrapping existing tensors

Parameters

- **images** – Tensor
- **targets** – Tensor

labels() → Optional[numpy.ndarray]

CHAPTER

TEN

GENERATORS

10.1 Generators

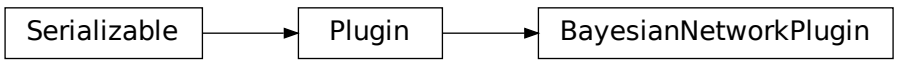
10.1.1 General purpose

synthcity.plugins.generic.plugin_bayesian_network module

Reference: Jankan, Ankur and Panda, Abinash, "pgmpy: Probabilistic graphical models using python," Proceedings of the 14th Python in Science Conference (SCIPY 2015), 2015.

class BayesianNetworkPlugin(*struct_learning_n_iter: int = 1000*, *struct_learning_search_method: str = 'tree_search'*, *struct_learning_score: str = 'k2'*, *struct_max_indegree: int = 4*, *encoder_max_clusters: int = 10*, *encoder_noise_scale: float = 0.1*, *workspace: pathlib.Path = PosixPath('workspace')*, *compress_dataset: bool = False*, *random_state: int = 0*, *sampling_patience: int = 500*, ***kwargs: Any*)

Bases: *synthcity.plugins.core.plugin.Plugin*



Bayesian Network for generative modeling. Implemented using pgmpy backend. :param struct_learning_n_iter: int

Number of iterations for the DAG learning

Parameters

- **struct_learning_search_method** – str = "tree_search" Search method for learning the DAG: hillclimb, pc, tree_search, mmhc, exhaustive
- **struct_learning_score** – str = "k2", Scoring for the DAG search: k2, bdeu, bic, bds
- **struct_max_indegree** – int = 4 The maximum number of parents for each node.
- **encoder_max_clusters** – int = 10 Data encoding clusters.
- **encoder_noise_scale** – float. Small noise to add to the final data, to prevent data leakage.

- **workspace** – Path. Optional Path for caching intermediary results.
- **compress_dataset** – bool. Default = False. Drop redundant features before training the generator.
- **random_state** – int. Random seed.
- **sampling_patience** – int. Max inference iterations to wait for the generated data to match the training schema.

Example

```
>>> from sklearn.datasets import load_iris
>>> from synthcity.plugins import Plugins
>>>
>>> X, y = load_iris(as_frame = True, return_X_y = True)
>>> X["target"] = y
>>>
>>> plugin = Plugins().get("bayesian_network")
>>> plugin.fit(X)
>>>
>>> plugin.generate(50)
```

class Config

Bases: object

arbitrary_types_allowed = True

validate_assignment = True

fit(*X: Union[*synthcity.plugins.core.dataloader.DataLoader*, pandas.core.frame.DataFrame]*, **args: Any*, ***kwargs: Any*) → Any

Training method the synthetic data plugin.

Parameters

- **X** – DataLoader. The reference dataset.
- **cond** – Optional, Union[pd.DataFrame, pd.Series, np.ndarray] Optional Training Conditional. The training conditional can be used to control to output of some models, like GANs or VAEs. The content can be anything, as long as it maps to the training dataset X. Usage example:

```
>>> from sklearn.datasets import load_iris
>>> from synthcity.plugins.core.dataloader import␣
↪GenericDataLoader
>>> from synthcity.plugins.core.constraints import Constraints
>>>
>>> # Load in `test_plugin` the generative model of choice
>>> # ....
>>>
>>> X, y = load_iris(as_frame=True, return_X_y=True)
>>> X["target"] = y
>>>
>>> X = GenericDataLoader(X)
>>> test_plugin.fit(X, cond=y)
>>>
```

(continues on next page)

(continued from previous page)

```
>>> count = 10
>>> X_gen = test_plugin.generate(count, cond=np.ones(count))
>>>
>>> # The Conditional only optimizes the output generation
>>> # for GANs and VAEs, but does NOT guarantee the samples
>>> # are only from that condition.
>>> # If you want to guarantee that output contains only
>>> # "target" == 1 samples, use Constraints.
>>>
>>> constraints = Constraints(
>>>     rules=[
>>>         ("target", "==", 1),
>>>     ]
>>> )
>>> X_gen = test_plugin.generate(count,
>>>         cond=np.ones(count),
>>>         constraints=constraints
>>>        )
>>> assert (X_gen["target"] == 1).all()
```

Returns self

classmethod fqdn() → str
The Fully-Qualified name of the plugin.

generate(*count: Optional[int] = None*, *constraints: Optional[*synthcity.plugins.core.constraints.Constraints*] = None*, *random_state: Optional[int] = None*, ***kwargs: Any*) → *synthcity.plugins.core.dataloader.DataLoader*
Synthetic data generation method.

Parameters

- **count** – optional int. The number of samples to generate. If None, it generated len(reference_dataset) samples.
- **cond** – Optional, Union[pd.DataFrame, pd.Series, np.ndarray]. Optional Generation Conditional. The conditional can be used only if the model was trained using a conditional too. If provided, it must have *count* length. Not all models support conditionals. The conditionals can be used in VAEs or GANs to speed-up the generation under some constraints. For model agnostic solutions, check out the *constraints* parameter.
- **constraints** – optional Constraints. Optional constraints to apply on the generated data. If none, the reference schema constraints are applied. The constraints are model agnostic, and will filter the output of the generative model. The constraints are a list of rules. Each rule is a tuple of the form (<feature>, <operation>, <value>).

  Valid Operations:

  - ”<”, “lt” : less than <value>
  - ”<=”, “le”: less or equal with <value>
  - ”>”, “gt” : greater than <value>
  - ”>=”, “ge”: greater or equal with <value>
  - ”==”, “eq”: equal with <value>

- ”in”: valid for categorical features, and <value> must be array. for example, (“target”, “in”, [0, 1])
- ”dtype”: <value> can be a data type. For example, (“target”, “dtype”, “int”)

Usage example:

```
>>> from synthcity.plugins.core.constraints import Constraints
>>> constraints = Constraints(
>>>   rules=[
>>>             ("InterestingFeature", "==", 0),
>>>         ]
>>>     )
>>>
>>> syn_data = syn_model.generate(
        count=count,
        constraints=constraints
    ).dataframe()
>>>
>>> assert (syn_data["InterestingFeature"] == 0).all()
```

- **random_state** – optional int. Optional random seed to use.

Returns <count> synthetic samples

static hyperparameter_space(***kwargs: Any*) → List[*synthcity.plugins.core.distribution.Distribution*]
Returns the hyperparameter space for the derived plugin.

static load(*buff: bytes*) → Any

static load_dict(*representation: dict*) → Any

static name() → str
The name of the plugin.

plot(*plt: Any*, *X:* synthcity.plugins.core.dataloader.DataLoader, *count: Optional[int] = None*, *plots: list = ['marginal', 'associations', 'tsne']*, ***kwargs: Any*) → Any
Plot the real-synthetic distributions.

Parameters

- **plt** – output
- **X** – DataLoader. The reference dataset.

Returns self

classmethod sample_hyperparameters(**args: Any*, ***kwargs: Any*) → Dict[str, Any]
Sample value from the hyperparameter space for the current plugin.

classmethod sample_hyperparameters_optuna(*trial: Any*, **args: Any*, ***kwargs: Any*) → Dict[str, Any]

save() → bytes

save_dict() → dict

save_to_file(*path: pathlib.Path*) → bytes

schema() → *synthcity.plugins.core.schema.Schema*
The reference schema

schema_includes(*other: Union[synthcity.plugins.core.dataloader.DataLoader, pandas.core.frame.DataFrame]*) → bool
Helper method to test if the reference schema includes a Dataset

Parameters **other** – DataLoader. The dataset to test

Returns bool, if the schema includes the dataset or not.

training_schema() → *synthcity.plugins.core.schema.Schema*
The internal schema

static type() → str
The type of the plugin.

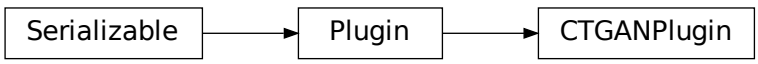
static version() → str
API version

plugin
alias of *synthcity.plugins.generic.plugin_bayesian_network.BayesianNetworkPlugin*

synthcity.plugins.generic.plugin_ctgan module

Reference: "Modeling Tabular Data using Conditional GAN", Xu, Lei et al.

class CTGANPlugin(*n_iter: int = 2000, generator_n_layers_hidden: int = 2, generator_n_units_hidden: int = 500, generator_nonlin: str = 'relu', generator_dropout: float = 0.1, generator_opt_betas: tuple = (0.5, 0.999), discriminator_n_layers_hidden: int = 2, discriminator_n_units_hidden: int = 500, discriminator_nonlin: str = 'leaky_relu', discriminator_n_iter: int = 1, discriminator_dropout: float = 0.1, discriminator_opt_betas: tuple = (0.5, 0.999), lr: float = 0.001, weight_decay: float = 0.001, batch_size: int = 200, random_state: int = 0, clipping_value: int = 1, lambda_gradient_penalty: float = 10, encoder_max_clusters: int = 10, encoder: Any = None, dataloader_sampler: Optional[torch.utils.data.sampler.Sampler] = None, device: Any = device(type='cpu'), patience: int = 5, patience_metric: Optional[synthcity.metrics.weighted_metrics.WeightedMetrics] = None, n_iter_print: int = 50, n_iter_min: int = 100, adjust_inference_sampling: bool = False, workspace: pathlib.Path = PosixPath('workspace'), compress_dataset: bool = False, sampling_patience: int = 500, **kwargs: Any*)
Bases: *synthcity.plugins.core.plugin.Plugin*



Conditional Tabular GAN implementation.

Parameters

- **generator_n_layers_hidden** – int Number of hidden layers in the generator
- **generator_n_units_hidden** – int Number of hidden units in each layer of the Generator
- **generator_nonlin** – string, default 'leaky_relu' Nonlinearity to use in the generator. Can be 'elu', 'relu', 'selu' or 'leaky_relu'.

- **n_iter** – int Maximum number of iterations in the Generator.
- **generator_dropout** – float Dropout value. If 0, the dropout is not used.
- **discriminator_n_layers_hidden** – int Number of hidden layers in the discriminator
- **discriminator_n_units_hidden** – int Number of hidden units in each layer of the discriminator
- **discriminator_nonlin** – string, default 'leaky_relu' Nonlinearity to use in the discriminator. Can be 'elu', 'relu', 'selu' or 'leaky_relu'.
- **discriminator_n_iter** – int Maximum number of iterations in the discriminator.
- **discriminator_dropout** – float Dropout value for the discriminator. If 0, the dropout is not used.
- **lr** – float learning rate for optimizer.
- **weight_decay** – float l2 (ridge) penalty for the weights.
- **batch_size** – int Batch size
- **random_state** – int random seed to use
- **clipping_value** – int, default 0 Gradients clipping value. Zero disables the feature
- **encoder_max_clusters** – int The max number of clusters to create for continuous columns when encoding
- **adjust_inference_sampling** – bool Adjust the marginal probabilities in the synthetic data to closer match the training set. Active only with the ConditionalSampler
- **stopping** (*# early*) –
- **n_iter_print** – int Number of iterations after which to print updates and check the validation loss.
- **n_iter_min** – int Minimum number of iterations to go through before starting early stopping
- **patience** – int Max number of iterations without any improvement before early stopping is trigged.
- **patience_metric** – Optional[WeightedMetrics] If not None, the metric is used for evaluation the criterion for early stopping.
- **arguments** (*# Core Plugin*) –
- **workspace** – Path. Optional Path for caching intermediary results.
- **compress_dataset** – bool. Default = False. Drop redundant features before training the generator.
- **sampling_patience** – int. Max inference iterations to wait for the generated data to match the training schema.

Example

```
>>> from sklearn.datasets import load_iris
>>> from synthcity.plugins import Plugins
>>>
>>> X, y = load_iris(as_frame = True, return_X_y = True)
>>> X["target"] = y
>>>
>>> plugin = Plugins().get("ctgan", n_iter = 100)
>>> plugin.fit(X)
>>>
>>> plugin.generate(50)
```

class Config

Bases: object

arbitrary_types_allowed = True

validate_assignment = True

fit(*X: Union[*synthcity.plugins.core.dataloader.DataLoader*, pandas.core.frame.DataFrame]*, **args: Any*, ***kwargs: Any*) → Any

Training method the synthetic data plugin.

Parameters

- **X** – DataLoader. The reference dataset.
- **cond** – Optional, Union[pd.DataFrame, pd.Series, np.ndarray] Optional Training Conditional. The training conditional can be used to control to output of some models, like GANs or VAEs. The content can be anything, as long as it maps to the training dataset X. Usage example:

```
>>> from sklearn.datasets import load_iris
>>> from synthcity.plugins.core.dataloader import␣
↪GenericDataLoader
>>> from synthcity.plugins.core.constraints import Constraints
>>>
>>> # Load in `test_plugin` the generative model of choice
>>> # ....
>>>
>>> X, y = load_iris(as_frame=True, return_X_y=True)
>>> X["target"] = y
>>>
>>> X = GenericDataLoader(X)
>>> test_plugin.fit(X, cond=y)
>>>
>>> count = 10
>>> X_gen = test_plugin.generate(count, cond=np.ones(count))
>>>
>>> # The Conditional only optimizes the output generation
>>> # for GANs and VAEs, but does NOT guarantee the samples
>>> # are only from that condition.
>>> # If you want to guarantee that output contains only
>>> # "target" == 1 samples, use Constraints.
>>>
```

(continues on next page)

(continued from previous page)

```
>>> constraints = Constraints(
>>>     rules=[
>>>         ("target", "==", 1),
>>>     ]
>>> )
>>> X_gen = test_plugin.generate(count,
>>>         cond=np.ones(count),
>>>         constraints=constraints
>>>        )
>>> assert (X_gen["target"] == 1).all()
```

Returns self

classmethod fqdn() → str
The Fully-Qualified name of the plugin.

generate(*count: Optional[int] = None*, *constraints: Optional[*synthcity.plugins.core.constraints.Constraints*] = None*, *random_state: Optional[int] = None*, ***kwargs: Any*) → *synthcity.plugins.core.dataloader.DataLoader*
Synthetic data generation method.

Parameters

- **count** – optional int. The number of samples to generate. If None, it generated len(reference_dataset) samples.
- **cond** – Optional, Union[pd.DataFrame, pd.Series, np.ndarray]. Optional Generation Conditional. The conditional can be used only if the model was trained using a conditional too. If provided, it must have *count* length. Not all models support conditionals. The conditionals can be used in VAEs or GANs to speed-up the generation under some constraints. For model agnostic solutions, check out the *constraints* parameter.
- **constraints** – optional Constraints. Optional constraints to apply on the generated data. If none, the reference schema constraints are applied. The constraints are model agnostic, and will filter the output of the generative model. The constraints are a list of rules. Each rule is a tuple of the form (<feature>, <operation>, <value>).

  Valid Operations:

  – ”<”, “lt” : less than <value>
  – ”<=”, “le”: less or equal with <value>
  – ”>”, “gt” : greater than <value>
  – ”>=”, “ge”: greater or equal with <value>
  – ”==”, “eq”: equal with <value>
  – ”in”: valid for categorical features, and <value> must be array. for example, (“target”, “in”, [0, 1])
  – ”dtype”: <value> can be a data type. For example, (“target”, “dtype”, “int”)

  Usage example:

```
>>> from synthcity.plugins.core.constraints import Constraints
>>> constraints = Constraints(
>>>   rules=[
>>>             ("InterestingFeature", "==", 0),
```

(continues on next page)

(continued from previous page)

```
>>>         ]
>>>     )
>>>
>>> syn_data = syn_model.generate(
        count=count,
        constraints=constraints
    ).dataframe()
>>>
>>> assert (syn_data["InterestingFeature"] == 0).all()
```

- **random_state** – optional int. Optional random seed to use.

Returns <count> synthetic samples

static hyperparameter_space(***kwargs: Any*) → List[*synthcity.plugins.core.distribution.Distribution*]
Returns the hyperparameter space for the derived plugin.

static load(*buff: bytes*) → Any

static load_dict(*representation: dict*) → Any

static name() → str
The name of the plugin.

plot(*plt: Any*, *X:* synthcity.plugins.core.dataloader.DataLoader, *count: Optional[int] = None*, *plots: list = ['marginal', 'associations', 'tsne']*, ***kwargs: Any*) → Any
Plot the real-synthetic distributions.

Parameters

- **plt** – output
- **X** – DataLoader. The reference dataset.

Returns self

classmethod sample_hyperparameters(**args: Any*, ***kwargs: Any*) → Dict[str, Any]
Sample value from the hyperparameter space for the current plugin.

classmethod sample_hyperparameters_optuna(*trial: Any*, **args: Any*, ***kwargs: Any*) → Dict[str, Any]

save() → bytes

save_dict() → dict

save_to_file(*path: pathlib.Path*) → bytes

schema() → *synthcity.plugins.core.schema.Schema*
The reference schema

schema_includes(*other: Union[*synthcity.plugins.core.dataloader.DataLoader, *pandas.core.frame.DataFrame]*) → bool
Helper method to test if the reference schema includes a Dataset

Parameters **other** – DataLoader. The dataset to test

Returns bool, if the schema includes the dataset or not.

training_schema() → *synthcity.plugins.core.schema.Schema*
The internal schema

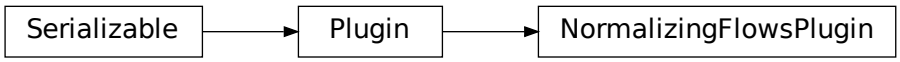
static type() → str
The type of the plugin.

static version() → str
API version

plugin
alias of *synthcity.plugins.generic.plugin_ctgan.CTGANPlugin*

synthcity.plugins.generic.plugin_nflow module

class NormalizingFlowsPlugin(*n_iter: int = 1000*, *n_layers_hidden: int = 1*, *n_units_hidden: int = 100*, *batch_size: int = 200*, *num_transform_blocks: int = 1*, *dropout: float = 0.1*, *batch_norm: bool = False*, *num_bins: int = 8*, *tail_bound: float = 3*, *lr: float = 0.001*, *apply_unconditional_transform: bool = True*, *base_distribution: str = 'standard_normal'*, *linear_transform_type: str = 'permutation'*, *base_transform_type: str = 'rq-autoregressive'*, *encoder_max_clusters: int = 10*, *tabular: bool = True*, *n_iter_min: int = 100*, *n_iter_print: int = 50*, *patience: int = 5*, *patience_metric: Optional[synthcity.metrics.weighted_metrics.WeightedMetrics] = None*, *workspace: pathlib.Path = PosixPath('workspace')*, *compress_dataset: bool = False*, *sampling_patience: int = 500*, *random_state: int = 0*, *device: Any = device(type='cpu')*, ***kwargs: Any*)
Bases: *synthcity.plugins.core.plugin.Plugin*



Normalizing Flows methods.

Normalizing Flows are generative models which produce tractable distributions where both sampling and density evaluation can be efficient and exact.

Parameters

- **n_iter** – int Number of flow steps
- **n_layers_hidden** – int Number of transformation layers
- **n_units_hidden** – int Number of hidden units for each layer
- **batch_size** – int Size of batch used for training
- **num_transform_blocks** – int Number of blocks to use in coupling/autoregressive nets.
- **dropout** – float Dropout probability for coupling/autoregressive nets.
- **batch_norm** – bool Whether to use batch norm in coupling/autoregressive nets.
- **num_bins** – int Number of bins to use for piecewise transforms.
- **tail_bound** – float Box is on [-bound, bound]^2
- **lr** – float Learning rate for optimizer.

- **apply_unconditional_transform** – bool Whether to unconditionally transform 'identity' features in the coupling layer.
- **base_distribution** – str Possible values: "standard_normal"
- **linear_transform_type** – str Type of linear transform to use. Possible values:
  - lu : A linear transform where we parameterize the LU decomposition of the weights.
  - permutation: Permutes using a random, but fixed, permutation.
  - svd: A linear module using the SVD decomposition for the weight matrix.
- **base_transform_type** – str Type of transform to use between linear layers. Possible values:
  - **affine-coupling** [An affine coupling layer that scales and shifts part of the variables.] Ref: L. Dinh et al., "Density estimation using Real NVP".
  - **quadratic-coupling :** Ref: Müller et al., "Neural Importance Sampling".
  - **rq-coupling** [Rational Quadratic Coupling] Ref: Durkan et al, "Neural Spline Flows".
  - **affine-autoregressive :Affine Autoregressive Transform** Ref: Durkan et al, "Neural Spline Flows".
  - **quadratic-autoregressive** [Quadratic Autoregressive Transform] Ref: Durkan et al, "Neural Spline Flows".
  - **rq-autoregressive** [Rational Quadratic Autoregressive Transform] Ref: Durkan et al, "Neural Spline Flows".
- **stopping** (*# early*) –
- **n_iter_print** – int Number of iterations after which to print updates and check the validation loss.
- **n_iter_min** – int Minimum number of iterations to go through before starting early stopping
- **patience** – int Max number of iterations without any improvement before training early stopping is trigged.
- **patience_metric** – Optional[WeightedMetrics] If not None, the metric is used for evaluation the criterion for training early stopping.
- **arguments** (*# Core Plugin*) –
- **workspace** – Path. Optional Path for caching intermediary results.
- **compress_dataset** – bool. Default = False. Drop redundant features before training the generator.
- **sampling_patience** – int. Max inference iterations to wait for the generated data to match the training schema.
- **random_state** – int random seed to use

Example

```
>>> from sklearn.datasets import load_iris
>>> from synthcity.plugins import Plugins
>>>
>>> X, y = load_iris(as_frame = True, return_X_y = True)
>>> X["target"] = y
>>>
>>> plugin = Plugins().get("nflow", n_iter = 100)
>>> plugin.fit(X)
>>>
>>> plugin.generate(50)
```

class Config

Bases: object

arbitrary_types_allowed = True

validate_assignment = True

fit(*X: Union[*synthcity.plugins.core.dataloader.DataLoader*, pandas.core.frame.DataFrame]*, **args: Any*, ***kwargs: Any*) → Any

Training method the synthetic data plugin.

Parameters

- **X** – DataLoader. The reference dataset.
- **cond** – Optional, Union[pd.DataFrame, pd.Series, np.ndarray] Optional Training Conditional. The training conditional can be used to control to output of some models, like GANs or VAEs. The content can be anything, as long as it maps to the training dataset X. Usage example:

```
>>> from sklearn.datasets import load_iris
>>> from synthcity.plugins.core.dataloader import␣
↪GenericDataLoader
>>> from synthcity.plugins.core.constraints import Constraints
>>>
>>> # Load in `test_plugin` the generative model of choice
>>> # ....
>>>
>>> X, y = load_iris(as_frame=True, return_X_y=True)
>>> X["target"] = y
>>>
>>> X = GenericDataLoader(X)
>>> test_plugin.fit(X, cond=y)
>>>
>>> count = 10
>>> X_gen = test_plugin.generate(count, cond=np.ones(count))
>>>
>>> # The Conditional only optimizes the output generation
>>> # for GANs and VAEs, but does NOT guarantee the samples
>>> # are only from that condition.
>>> # If you want to guarantee that output contains only
>>> # "target" == 1 samples, use Constraints.
>>>
```

(continues on next page)

(continued from previous page)

```
>>> constraints = Constraints(
>>>     rules=[
>>>         ("target", "==", 1),
>>>     ]
>>> )
>>> X_gen = test_plugin.generate(count,
>>>         cond=np.ones(count),
>>>         constraints=constraints
>>>        )
>>> assert (X_gen["target"] == 1).all()
```

Returns self

classmethod fqdn() → str
The Fully-Qualified name of the plugin.

generate(*count: Optional[int] = None*, *constraints: Optional[*synthcity.plugins.core.constraints.Constraints*] = None*, *random_state: Optional[int] = None*, ***kwargs: Any*) → *synthcity.plugins.core.dataloader.DataLoader*
Synthetic data generation method.

Parameters

- **count** – optional int. The number of samples to generate. If None, it generated len(reference_dataset) samples.
- **cond** – Optional, Union[pd.DataFrame, pd.Series, np.ndarray]. Optional Generation Conditional. The conditional can be used only if the model was trained using a conditional too. If provided, it must have *count* length. Not all models support conditionals. The conditionals can be used in VAEs or GANs to speed-up the generation under some constraints. For model agnostic solutions, check out the *constraints* parameter.
- **constraints** – optional Constraints. Optional constraints to apply on the generated data. If none, the reference schema constraints are applied. The constraints are model agnostic, and will filter the output of the generative model. The constraints are a list of rules. Each rule is a tuple of the form (<feature>, <operation>, <value>).

  Valid Operations:

  - "<", "lt" : less than <value>
  - "<=", "le": less or equal with <value>
  - ">", "gt" : greater than <value>
  - ">=", "ge": greater or equal with <value>
  - "==", "eq": equal with <value>
  - "in": valid for categorical features, and <value> must be array. for example, ("target", "in", [0, 1])
  - "dtype": <value> can be a data type. For example, ("target", "dtype", "int")

  Usage example:

```
>>> from synthcity.plugins.core.constraints import Constraints
>>> constraints = Constraints(
>>>   rules=[
>>>             ("InterestingFeature", "==", 0),
```

(continues on next page)

(continued from previous page)

```
>>>         ]
>>>     )
>>>
>>> syn_data = syn_model.generate(
        count=count,
        constraints=constraints
    ).dataframe()
>>>
>>> assert (syn_data["InterestingFeature"] == 0).all()
```

- **random_state** – optional int. Optional random seed to use.

Returns <count> synthetic samples

static hyperparameter_space(***kwargs: Any*) → List[*synthcity.plugins.core.distribution.Distribution*]
Returns the hyperparameter space for the derived plugin.

static load(*buff: bytes*) → Any

static load_dict(*representation: dict*) → Any

static name() → str
The name of the plugin.

plot(*plt: Any*, *X:* synthcity.plugins.core.dataloader.DataLoader, *count: Optional[int] = None*, *plots: list = ['marginal', 'associations', 'tsne']*, ***kwargs: Any*) → Any
Plot the real-synthetic distributions.

Parameters

- **plt** – output
- **X** – DataLoader. The reference dataset.

Returns self

classmethod sample_hyperparameters(**args: Any*, ***kwargs: Any*) → Dict[str, Any]
Sample value from the hyperparameter space for the current plugin.

classmethod sample_hyperparameters_optuna(*trial: Any*, **args: Any*, ***kwargs: Any*) → Dict[str, Any]

save() → bytes

save_dict() → dict

save_to_file(*path: pathlib.Path*) → bytes

schema() → *synthcity.plugins.core.schema.Schema*
The reference schema

schema_includes(*other: Union[*synthcity.plugins.core.dataloader.DataLoader, *pandas.core.frame.DataFrame]*) → bool
Helper method to test if the reference schema includes a Dataset

Parameters **other** – DataLoader. The dataset to test

Returns bool, if the schema includes the dataset or not.

training_schema() → *synthcity.plugins.core.schema.Schema*
The internal schema

static type() → str
The type of the plugin.

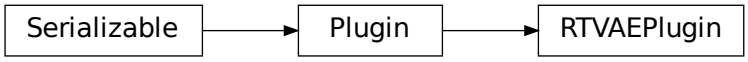
static version() → str
API version

plugin
alias of *synthcity.plugins.generic.plugin_nflow.NormalizingFlowsPlugin*

synthcity.plugins.generic.plugin_rtvae module

Reference: Akrami, Haleh, Anand A. Joshi, Jian Li, Sergül Aydöre, and Richard M. Leahy. "A robust variational autoencoder using beta divergence." Knowledge-Based Systems 238 (2022): 107886.

class RTVAEPlugin(*n_iter: int = 1000*, *n_units_embedding: int = 500*, *lr: float = 0.001*, *weight_decay: float = 1e-05*, *batch_size: int = 200*, *random_state: int = 0*, *decoder_n_layers_hidden: int = 3*, *decoder_n_units_hidden: int = 500*, *decoder_nonlin: str = 'leaky_relu'*, *decoder_dropout: float = 0*, *encoder_n_layers_hidden: int = 3*, *encoder_n_units_hidden: int = 500*, *encoder_nonlin: str = 'leaky_relu'*, *encoder_dropout: float = 0.1*, *data_encoder_max_clusters: int = 10*, *robust_divergence_beta: int = 2*, *dataloader_sampler: Optional[torch.utils.data.sampler.Sampler] = None*, *n_iter_print: int = 50*, *n_iter_min: int = 100*, *patience: int = 5*, *device: Any = device(type='cpu')*, *workspace: pathlib.Path = PosixPath('workspace')*, *compress_dataset: bool = False*, *sampling_patience: int = 500*, ***kwargs: Any*)
Bases: *synthcity.plugins.core.plugin.Plugin*



Tabular VAE with robust beta divergence.

Parameters

- **decoder_n_layers_hidden** – int Number of hidden layers in the decoder
- **decoder_n_units_hidden** – int Number of hidden units in each layer of the decoder
- **decoder_nonlin** – string, default 'leaky_relu' Nonlinearity to use in the decoder. Can be 'elu', 'relu', 'selu' or 'leaky_relu'.
- **decoder_dropout** – float Dropout value. If 0, the dropout is not used.
- **encoder_n_layers_hidden** – int Number of hidden layers in the encoder
- **encoder_n_units_hidden** – int Number of hidden units in each layer of the encoder
- **encoder_nonlin** – string, default 'leaky_relu' Nonlinearity to use in the encoder. Can be 'elu', 'relu', 'selu' or 'leaky_relu'.
- **encoder_dropout** – float Dropout value for the encoder. If 0, the dropout is not used.
- **n_iter** – int Maximum number of iterations in the encoder.
- **lr** – float learning rate for optimizer.

- **weight_decay** – float l2 (ridge) penalty for the weights.
- **batch_size** – int Batch size
- **random_state** – int random_state used
- **encoder_max_clusters** – int The max number of clusters to create for continuous columns when encoding
- **arguments** (*# Core Plugin*) –
- **workspace** – Path. Optional Path for caching intermediary results.
- **compress_dataset** – bool. Default = False. Drop redundant features before training the generator.
- **sampling_patience** – int. Max inference iterations to wait for the generated data to match the training schema.

Example

```
>>> from sklearn.datasets import load_iris
>>> from synthcity.plugins import Plugins
>>>
>>> X, y = load_iris(as_frame = True, return_X_y = True)
>>> X["target"] = y
>>>
>>> plugin = Plugins().get("rtvae", n_iter = 100)
>>> plugin.fit(X)
>>>
>>> plugin.generate(50)
```

class Config

Bases: object

arbitrary_types_allowed = True

validate_assignment = True

fit(*X: Union[*synthcity.plugins.core.dataloader.DataLoader*, pandas.core.frame.DataFrame]*, **args: Any*, ***kwargs: Any*) → Any

Training method the synthetic data plugin.

Parameters

- **X** – DataLoader. The reference dataset.
- **cond** – Optional, Union[pd.DataFrame, pd.Series, np.ndarray] Optional Training Conditional. The training conditional can be used to control to output of some models, like GANs or VAEs. The content can be anything, as long as it maps to the training dataset X. Usage example:

```
>>> from sklearn.datasets import load_iris
>>> from synthcity.plugins.core.dataloader import␣
↪GenericDataLoader
>>> from synthcity.plugins.core.constraints import Constraints
>>>
>>> # Load in `test_plugin` the generative model of choice
>>> # ....
```

(continues on next page)

(continued from previous page)

```
>>>
>>> X, y = load_iris(as_frame=True, return_X_y=True)
>>> X["target"] = y
>>>
>>> X = GenericDataLoader(X)
>>> test_plugin.fit(X, cond=y)
>>>
>>> count = 10
>>> X_gen = test_plugin.generate(count, cond=np.ones(count))
>>>
>>> # The Conditional only optimizes the output generation
>>> # for GANs and VAEs, but does NOT guarantee the samples
>>> # are only from that condition.
>>> # If you want to guarantee that output contains only
>>> # "target" == 1 samples, use Constraints.
>>>
>>> constraints = Constraints(
>>>     rules=[
>>>         ("target", "==", 1),
>>>     ]
>>> )
>>> X_gen = test_plugin.generate(count,
>>>         cond=np.ones(count),
>>>         constraints=constraints
>>>        )
>>> assert (X_gen["target"] == 1).all()
```

Returns self

classmethod fqdn() → str
The Fully-Qualified name of the plugin.

generate(*count: Optional[int] = None*, *constraints: Optional[*synthcity.plugins.core.constraints.Constraints*] = None*, *random_state: Optional[int] = None*, ***kwargs: Any*) → *synthcity.plugins.core.dataloader.DataLoader*
Synthetic data generation method.

Parameters

- **count** – optional int. The number of samples to generate. If None, it generated len(reference_dataset) samples.
- **cond** – Optional, Union[pd.DataFrame, pd.Series, np.ndarray]. Optional Generation Conditional. The conditional can be used only if the model was trained using a conditional too. If provided, it must have *count* length. Not all models support conditionals. The conditionals can be used in VAEs or GANs to speed-up the generation under some constraints. For model agnostic solutions, check out the *constraints* parameter.
- **constraints** – optional Constraints. Optional constraints to apply on the generated data. If none, the reference schema constraints are applied. The constraints are model agnostic, and will filter the output of the generative model. The constraints are a list of rules. Each rule is a tuple of the form (<feature>, <operation>, <value>).

  Valid Operations:

  – ”<”, “lt” : less than <value>

- "<=", "le": less or equal with <value>
- ">", "gt" : greater than <value>
- ">=", "ge": greater or equal with <value>
- "==", "eq": equal with <value>
- "in": valid for categorical features, and <value> must be array. for example, ("target", "in", [0, 1])
- "dtype": <value> can be a data type. For example, ("target", "dtype", "int")

Usage example:

```
>>> from synthcity.plugins.core.constraints import Constraints
>>> constraints = Constraints(
>>>   rules=[
>>>             ("InterestingFeature", "==", 0),
>>>         ]
>>>     )
>>>
>>> syn_data = syn_model.generate(
        count=count,
        constraints=constraints
    ).dataframe()
>>>
>>> assert (syn_data["InterestingFeature"] == 0).all()
```

- **random_state** – optional int. Optional random seed to use.

Returns <count> synthetic samples

static hyperparameter_space(***kwargs: Any*) → List[*synthcity.plugins.core.distribution.Distribution*]
Returns the hyperparameter space for the derived plugin.

static load(*buff: bytes*) → Any

static load_dict(*representation: dict*) → Any

static name() → str
The name of the plugin.

plot(*plt: Any*, *X:* synthcity.plugins.core.dataloader.DataLoader, *count: Optional[int] = None*, *plots: list = ['marginal', 'associations', 'tsne']*, ***kwargs: Any*) → Any
Plot the real-synthetic distributions.

Parameters

- **plt** – output
- **X** – DataLoader. The reference dataset.

Returns self

classmethod sample_hyperparameters(**args: Any*, ***kwargs: Any*) → Dict[str, Any]
Sample value from the hyperparameter space for the current plugin.

classmethod sample_hyperparameters_optuna(*trial: Any*, **args: Any*, ***kwargs: Any*) → Dict[str, Any]

save() → bytes

save_dict() → dict

save_to_file(*path: pathlib.Path*) → bytes

schema() → *synthcity.plugins.core.schema.Schema*
The reference schema

schema_includes(*other: Union[*synthcity.plugins.core.dataloader.DataLoader, *pandas.core.frame.DataFrame]*) → bool
Helper method to test if the reference schema includes a Dataset

Parameters **other** – DataLoader. The dataset to test

Returns bool, if the schema includes the dataset or not.

training_schema() → *synthcity.plugins.core.schema.Schema*
The internal schema

static type() → str
The type of the plugin.

static version() → str
API version

plugin
alias of *synthcity.plugins.generic.plugin_rtvae.RTVAEPlugin*

synthcity.plugins.generic.plugin_tvae module

class TVAEPlugin(*n_iter: int = 1000*, *n_units_embedding: int = 500*, *lr: float = 0.001*, *weight_decay: float = 1e-05*, *batch_size: int = 200*, *random_state: int = 0*, *decoder_n_layers_hidden: int = 3*, *decoder_n_units_hidden: int = 500*, *decoder_nonlin: str = 'leaky_relu'*, *decoder_dropout: float = 0*, *encoder_n_layers_hidden: int = 3*, *encoder_n_units_hidden: int = 500*, *encoder_nonlin: str = 'leaky_relu'*, *encoder_dropout: float = 0.1*, *loss_factor: int = 1*, *data_encoder_max_clusters: int = 10*, *dataloader_sampler: Optional[torch.utils.data.sampler.Sampler] = None*, *clipping_value: int = 1*, *n_iter_print: int = 50*, *n_iter_min: int = 100*, *patience: int = 5*, *device: Any = device(type='cpu')*, *workspace: pathlib.Path = PosixPath('workspace')*, *compress_dataset: bool = False*, *sampling_patience: int = 500*, ***kwargs: Any*)
Bases: *synthcity.plugins.core.plugin.Plugin*

Tabular VAE implementation.

Parameters

- **decoder_n_layers_hidden** – int Number of hidden layers in the decoder
- **decoder_n_units_hidden** – int Number of hidden units in each layer of the decoder
- **decoder_nonlin** – string, default ‘leaky_relu’ Nonlinearity to use in the decoder. Can be ‘elu’, ‘relu’, ‘selu’ or ‘leaky_relu’.
- **decoder_dropout** – float Dropout value. If 0, the dropout is not used.

- **encoder_n_layers_hidden** – int Number of hidden layers in the encoder
- **encoder_n_units_hidden** – int Number of hidden units in each layer of the encoder
- **encoder_nonlin** – string, default 'leaky_relu' Nonlinearity to use in the encoder. Can be 'elu', 'relu', 'selu' or 'leaky_relu'.
- **encoder_dropout** – float Dropout value for the encoder. If 0, the dropout is not used.
- **n_iter** – int Maximum number of iterations in the encoder.
- **lr** – float learning rate for optimizer.
- **weight_decay** – float l2 (ridge) penalty for the weights.
- **batch_size** – int Batch size
- **random_state** – int random_state used
- **encoder_max_clusters** – int The max number of clusters to create for continuous columns when encoding
- **stopping** (*# early*) –
- **n_iter_print** – int Number of iterations after which to print updates and check the validation loss.
- **n_iter_min** – int Minimum number of iterations to go through before starting early stopping
- **patience** – int Max number of iterations without any improvement before early stopping is trigged.
- **arguments** (*# Core Plugin*) –
- **workspace** – Path. Optional Path for caching intermediary results.
- **compress_dataset** – bool. Default = False. Drop redundant features before training the generator.
- **sampling_patience** – int. Max inference iterations to wait for the generated data to match the training schema.

Example

```
>>> from sklearn.datasets import load_iris
>>> from synthcity.plugins import Plugins
>>>
>>> X, y = load_iris(as_frame = True, return_X_y = True)
>>> X["target"] = y
>>>
>>> plugin = Plugins().get("tvae", n_iter = 100)
>>> plugin.fit(X)
>>>
>>> plugin.generate(50)
```

class Config

Bases: object

arbitrary_types_allowed = True

validate_assignment = True

fit(*X: Union[synthcity.plugins.core.dataloader.DataLoader, pandas.core.frame.DataFrame]*, **args: Any*, ***kwargs: Any*) → Any
Training method the synthetic data plugin.

Parameters

- **X** – DataLoader. The reference dataset.
- **cond** – Optional, Union[pd.DataFrame, pd.Series, np.ndarray] Optional Training Conditional. The training conditional can be used to control to output of some models, like GANs or VAEs. The content can be anything, as long as it maps to the training dataset X. Usage example:

```
>>> from sklearn.datasets import load_iris
>>> from synthcity.plugins.core.dataloader import␣
↪GenericDataLoader
>>> from synthcity.plugins.core.constraints import Constraints
>>>
>>> # Load in `test_plugin` the generative model of choice
>>> # ....
>>>
>>> X, y = load_iris(as_frame=True, return_X_y=True)
>>> X["target"] = y
>>>
>>> X = GenericDataLoader(X)
>>> test_plugin.fit(X, cond=y)
>>>
>>> count = 10
>>> X_gen = test_plugin.generate(count, cond=np.ones(count))
>>>
>>> # The Conditional only optimizes the output generation
>>> # for GANs and VAEs, but does NOT guarantee the samples
>>> # are only from that condition.
>>> # If you want to guarantee that output contains only
>>> # "target" == 1 samples, use Constraints.
>>>
>>> constraints = Constraints(
>>>     rules=[
>>>         ("target", "==", 1),
>>>     ]
>>> )
>>> X_gen = test_plugin.generate(count,
>>>         cond=np.ones(count),
>>>         constraints=constraints
>>>        )
>>> assert (X_gen["target"] == 1).all()
```

Returns self

classmethod fqdn() → str
The Fully-Qualified name of the plugin.

generate(*count: Optional[int] = None*, *constraints: Optional[synthcity.plugins.core.constraints.Constraints] = None*, *random_state: Optional[int] = None*, ***kwargs: Any*) → *synthcity.plugins.core.dataloader.DataLoader*
Synthetic data generation method.

Parameters

- **count** – optional int. The number of samples to generate. If None, it generated len(reference_dataset) samples.
- **cond** – Optional, Union[pd.DataFrame, pd.Series, np.ndarray]. Optional Generation Conditional. The conditional can be used only if the model was trained using a conditional too. If provided, it must have *count* length. Not all models support conditionals. The conditionals can be used in VAEs or GANs to speed-up the generation under some constraints. For model agnostic solutions, check out the *constraints* parameter.
- **constraints** – optional Constraints. Optional constraints to apply on the generated data. If none, the reference schema constraints are applied. The constraints are model agnostic, and will filter the output of the generative model. The constraints are a list of rules. Each rule is a tuple of the form (<feature>, <operation>, <value>).

  Valid Operations:

  - "<", "lt" : less than <value>
  - "<=", "le": less or equal with <value>
  - ">", "gt" : greater than <value>
  - ">=", "ge": greater or equal with <value>
  - "==", "eq": equal with <value>
  - "in": valid for categorical features, and <value> must be array. for example, ("target", "in", [0, 1])
  - "dtype": <value> can be a data type. For example, ("target", "dtype", "int")

  Usage example:

```
>>> from synthcity.plugins.core.constraints import Constraints
>>> constraints = Constraints(
>>>   rules=[
>>>             ("InterestingFeature", "==", 0),
>>>         ]
>>>     )
>>>
>>> syn_data = syn_model.generate(
        count=count,
        constraints=constraints
    ).dataframe()
>>>
>>> assert (syn_data["InterestingFeature"] == 0).all()
```

- **random_state** – optional int. Optional random seed to use.

Returns <count> synthetic samples

static hyperparameter_space(***kwargs: Any*) → List[*synthcity.plugins.core.distribution.Distribution*]
Returns the hyperparameter space for the derived plugin.

static load(*buff: bytes*) → Any

static load_dict(*representation: dict*) → Any

static name() → str
The name of the plugin.

plot(*plt: Any*, *X:* synthcity.plugins.core.dataloader.DataLoader, *count: Optional[int] = None*, *plots: list = ['marginal', 'associations', 'tsne']*, ***kwargs: Any*) → Any
Plot the real-synthetic distributions.

Parameters

- **plt** – output
- **X** – DataLoader. The reference dataset.

Returns self

classmethod sample_hyperparameters(**args: Any*, ***kwargs: Any*) → Dict[str, Any]
Sample value from the hyperparameter space for the current plugin.

classmethod sample_hyperparameters_optuna(*trial: Any*, **args: Any*, ***kwargs: Any*) → Dict[str, Any]

save() → bytes

save_dict() → dict

save_to_file(*path: pathlib.Path*) → bytes

schema() → *synthcity.plugins.core.schema.Schema*
The reference schema

schema_includes(*other: Union[*synthcity.plugins.core.dataloader.DataLoader*, pandas.core.frame.DataFrame]*) → bool
Helper method to test if the reference schema includes a Dataset

Parameters **other** – DataLoader. The dataset to test

Returns bool, if the schema includes the dataset or not.

training_schema() → *synthcity.plugins.core.schema.Schema*
The internal schema

static type() → str
The type of the plugin.

static version() → str
API version

plugin
alias of *synthcity.plugins.generic.plugin_tvae.TVAEPlugin*

10.1.2 Privacy-focused

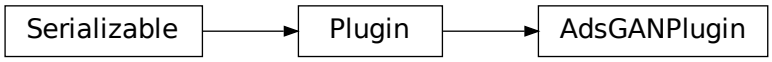
synthcity.plugins.privacy.plugin_adsgan module

Reference: Jinsung Yoon, Lydia N. Drumright, Mihaela van der Schaar, "Anonymization through Data Synthesis using Generative Adversarial Networks (ADS-GAN): A harmonizing advancement for AI in medicine," IEEE Journal of Biomedical and Health Informatics (JBHI), 2019.

Paper link: https://ieeexplore.ieee.org/document/9034117

class **AdsGANPlugin**(*n_iter: int = 10000*, *generator_n_layers_hidden: int = 2*, *generator_n_units_hidden: int = 500*, *generator_nonlin: str = 'relu'*, *generator_dropout: float = 0.1*, *generator_opt_betas: tuple = (0.5, 0.999)*, *discriminator_n_layers_hidden: int = 2*, *discriminator_n_units_hidden: int = 500*, *discriminator_nonlin: str = 'leaky_relu'*, *discriminator_n_iter: int = 1*, *discriminator_dropout: float = 0.1*, *discriminator_opt_betas: tuple = (0.5, 0.999)*, *lr: float = 0.001*, *weight_decay: float = 0.001*, *batch_size: int = 200*, *random_state: int = 0*, *clipping_value: int = 1*, *lambda_gradient_penalty: float = 10*, *lambda_identifiability_penalty: float = 0.1*, *encoder_max_clusters: int = 5*, *encoder: Any = None*, *dataloader_sampler: Optional[torch.utils.data.sampler.Sampler] = None*, *device: Any = device(type='cpu')*, *adjust_inference_sampling: bool = False*, *patience: int = 5*, *patience_metric: Optional[*synthcity.metrics.weighted_metrics.WeightedMetrics*] = None*, *n_iter_print: int = 50*, *n_iter_min: int = 100*, *workspace: pathlib.Path = PosixPath('workspace')*, *compress_dataset: bool = False*, *sampling_patience: int = 500*, ***kwargs: Any*)

Bases: *synthcity.plugins.core.plugin.Plugin*



AdsGAN plugin - Anonymization through Data Synthesis using Generative Adversarial Networks.

Parameters

- **generator_n_layers_hidden** – int Number of hidden layers in the generator
- **generator_n_units_hidden** – int Number of hidden units in each layer of the Generator
- **generator_nonlin** – string, default 'leaky_relu' Nonlinearity to use in the generator. Can be 'elu', 'relu', 'selu' or 'leaky_relu'.
- **n_iter** – int Maximum number of iterations in the Generator.
- **generator_dropout** – float Dropout value. If 0, the dropout is not used.
- **discriminator_n_layers_hidden** – int Number of hidden layers in the discriminator
- **discriminator_n_units_hidden** – int Number of hidden units in each layer of the discriminator
- **discriminator_nonlin** – string, default 'leaky_relu' Nonlinearity to use in the discriminator. Can be 'elu', 'relu', 'selu' or 'leaky_relu'.
- **discriminator_n_iter** – int Maximum number of iterations in the discriminator.
- **discriminator_dropout** – float Dropout value for the discriminator. If 0, the dropout is not used.
- **lr** – float learning rate for optimizer.
- **weight_decay** – float l2 (ridge) penalty for the weights.
- **batch_size** – int Batch size
- **random_state** – int random seed to use

- **clipping_value** – int, default 0 Gradients clipping value. Zero disables the feature
- **encoder_max_clusters** – int The max number of clusters to create for continuous columns when encoding
- **adjust_inference_sampling** – bool Adjust the marginal probabilities in the synthetic data to closer match the training set. Active only with the ConditionalSampler
- **lambda_gradient_penalty** – float = 10 Weight for the gradient penalty
- **lambda_identifiability_penalty** – float = 0.1 Weight for the identifiability penalty, if enabled
- **stopping** (*# early*) –
- **n_iter_print** – int Number of iterations after which to print updates and check the validation loss.
- **n_iter_min** – int Minimum number of iterations to go through before starting early stopping
- **patience** – int Max number of iterations without any improvement before training early stopping is trigged.
- **patience_metric** – Optional[WeightedMetrics] If not None, the metric is used for evaluation the criterion for training early stopping.
- **arguments** (*# Core Plugin*) –
- **workspace** – Path. Optional Path for caching intermediary results.
- **compress_dataset** – bool. Default = False. Drop redundant features before training the generator.
- **sampling_patience** – int. Max inference iterations to wait for the generated data to match the training schema.

Example

```
>>> from sklearn.datasets import load_iris
>>> from synthcity.plugins import Plugins
>>>
>>> X, y = load_iris(as_frame = True, return_X_y = True)
>>> X["target"] = y
>>>
>>> plugin = Plugins().get("adsgan", n_iter = 100)
>>> plugin.fit(X)
>>>
>>> plugin.generate(50)
```

class Config

Bases: object

arbitrary_types_allowed = True

validate_assignment = True

fit(*X: Union[*synthcity.plugins.core.dataloader.DataLoader*, pandas.core.frame.DataFrame]*, **args: Any*, ***kwargs: Any*) → Any

Training method the synthetic data plugin.

Parameters

- **X** – DataLoader. The reference dataset.
- **cond** – Optional, Union[pd.DataFrame, pd.Series, np.ndarray] Optional Training Conditional. The training conditional can be used to control to output of some models, like GANs or VAEs. The content can be anything, as long as it maps to the training dataset X. Usage example:

```
>>> from sklearn.datasets import load_iris
>>> from synthcity.plugins.core.dataloader import␣
↪GenericDataLoader
>>> from synthcity.plugins.core.constraints import Constraints
>>>
>>> # Load in `test_plugin` the generative model of choice
>>> # ....
>>>
>>> X, y = load_iris(as_frame=True, return_X_y=True)
>>> X["target"] = y
>>>
>>> X = GenericDataLoader(X)
>>> test_plugin.fit(X, cond=y)
>>>
>>> count = 10
>>> X_gen = test_plugin.generate(count, cond=np.ones(count))
>>>
>>> # The Conditional only optimizes the output generation
>>> # for GANs and VAEs, but does NOT guarantee the samples
>>> # are only from that condition.
>>> # If you want to guarantee that output contains only
>>> # "target" == 1 samples, use Constraints.
>>>
>>> constraints = Constraints(
>>>     rules=[
>>>         ("target", "==", 1),
>>>     ]
>>> )
>>> X_gen = test_plugin.generate(count,
>>>         cond=np.ones(count),
>>>         constraints=constraints
>>>        )
>>> assert (X_gen["target"] == 1).all()
```

Returns self

classmethod fqdn() → str
The Fully-Qualified name of the plugin.

generate(*count: Optional[int] = None*, *constraints: Optional[*synthcity.plugins.core.constraints.Constraints*] = None*, *random_state: Optional[int] = None*, ***kwargs: Any*) → *synthcity.plugins.core.dataloader.DataLoader*
Synthetic data generation method.

Parameters

- **count** – optional int. The number of samples to generate. If None, it generated len(reference_dataset) samples.

- **cond** – Optional, Union[pd.DataFrame, pd.Series, np.ndarray]. Optional Generation Conditional. The conditional can be used only if the model was trained using a conditional too. If provided, it must have *count* length. Not all models support conditionals. The conditionals can be used in VAEs or GANs to speed-up the generation under some constraints. For model agnostic solutions, check out the *constraints* parameter.
- **constraints** – optional Constraints. Optional constraints to apply on the generated data. If none, the reference schema constraints are applied. The constraints are model agnostic, and will filter the output of the generative model. The constraints are a list of rules. Each rule is a tuple of the form (<feature>, <operation>, <value>).

  Valid Operations:

  - "<", "lt" : less than <value>
  - "<=", "le": less or equal with <value>
  - ">", "gt" : greater than <value>
  - ">=", "ge": greater or equal with <value>
  - "==", "eq": equal with <value>
  - "in": valid for categorical features, and <value> must be array. for example, ("target", "in", [0, 1])
  - "dtype": <value> can be a data type. For example, ("target", "dtype", "int")

  Usage example:

```
>>> from synthcity.plugins.core.constraints import Constraints
>>> constraints = Constraints(
>>>   rules=[
>>>             ("InterestingFeature", "==", 0),
>>>         ]
>>>     )
>>>
>>> syn_data = syn_model.generate(
        count=count,
        constraints=constraints
    ).dataframe()
>>>
>>> assert (syn_data["InterestingFeature"] == 0).all()
```

- **random_state** – optional int. Optional random seed to use.

Returns <count> synthetic samples

static hyperparameter_space(***kwargs: Any*) → List[*synthcity.plugins.core.distribution.Distribution*]
Returns the hyperparameter space for the derived plugin.

static load(*buff: bytes*) → Any

static load_dict(*representation: dict*) → Any

static name() → str
The name of the plugin.

plot(*plt: Any*, *X:* synthcity.plugins.core.dataloader.DataLoader, *count: Optional[int] = None*, *plots: list = ['marginal', 'associations', 'tsne']*, ***kwargs: Any*) → Any
Plot the real-synthetic distributions.

Parameters

- **plt** – output
- **X** – DataLoader. The reference dataset.

Returns self

classmethod sample_hyperparameters(**args: Any*, ***kwargs: Any*) → Dict[str, Any]
Sample value from the hyperparameter space for the current plugin.

classmethod sample_hyperparameters_optuna(*trial: Any*, **args: Any*, ***kwargs: Any*) → Dict[str, Any]

save() → bytes

save_dict() → dict

save_to_file(*path: pathlib.Path*) → bytes

schema() → *synthcity.plugins.core.schema.Schema*
The reference schema

schema_includes(*other: Union[*synthcity.plugins.core.dataloader.DataLoader*, pandas.core.frame.DataFrame]*) → bool
Helper method to test if the reference schema includes a Dataset

Parameters **other** – DataLoader. The dataset to test

Returns bool, if the schema includes the dataset or not.

training_schema() → *synthcity.plugins.core.schema.Schema*
The internal schema

static type() → str
The type of the plugin.

static version() → str
API version

plugin
alias of *synthcity.plugins.privacy.plugin_adsgan.AdsGANPlugin*

synthcity.plugins.privacy.plugin_pategan module

Reference: James Jordon, Jinsung Yoon, Mihaela van der Schaar, "PATE-GAN: Generating Synthetic Data with Differential Privacy Guarantees," International Conference on Learning Representations (ICLR), 2019. Paper link: https://openreview.net/forum?id=S1zk9iRqF7

class PATEGAN(*max_iter: int = 1000*, *generator_n_layers_hidden: int = 2*, *generator_n_units_hidden: int = 100*, *generator_nonlin: str = 'relu'*, *generator_n_iter: int = 10*, *generator_dropout: float = 0*, *discriminator_n_layers_hidden: int = 2*, *discriminator_n_units_hidden: int = 100*, *discriminator_nonlin: str = 'leaky_relu'*, *discriminator_n_iter: int = 1*, *discriminator_dropout: float = 0.1*, *lr: float = 0.0001*, *weight_decay: float = 0.001*, *batch_size: int = 200*, *random_state: int = 0*, *clipping_value: int = 1*, *encoder_max_clusters: int = 5*, *device: Any = device(type='cpu')*, *n_teachers: int = 10*, *teacher_template: str = 'linear'*, *epsilon: float = 1.0*, *delta: Optional[float] = None*, *lamda: float = 0.001*, *alpha: int = 100*, *encoder: Any = None*)
Bases: *synthcity.plugins.core.serializable.Serializable*

Basic PATE-GAN framework.

fit(*X_train: pandas.core.frame.DataFrame*) → Any

static load(*buff: bytes*) → Any

static load_dict(*representation: dict*) → Any

sample(*count: int*) → numpy.ndarray

save() → bytes

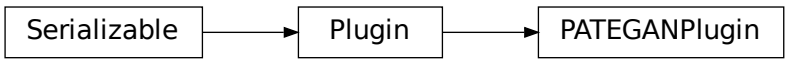
save_dict() → dict

save_to_file(*path: pathlib.Path*) → bytes

static version() → str
API version

class PATEGANPlugin(*n_iter: int = 200*, *generator_n_iter: int = 10*, *generator_n_layers_hidden: int = 2*, *generator_n_units_hidden: int = 500*, *generator_nonlin: str = 'relu'*, *generator_dropout: float = 0*, *discriminator_n_layers_hidden: int = 2*, *discriminator_n_units_hidden: int = 500*, *discriminator_nonlin: str = 'leaky_relu'*, *discriminator_n_iter: int = 1*, *discriminator_dropout: float = 0.1*, *lr: float = 0.001*, *weight_decay: float = 0.001*, *batch_size: int = 200*, *random_state: int = 0*, *clipping_value: int = 1*, *encoder_max_clusters: int = 5*, *n_teachers: int = 10*, *teacher_template: str = 'xgboost'*, *epsilon: float = 1.0*, *delta: Optional[float] = None*, *lamda: float = 0.001*, *alpha: int = 100*, *encoder: Any = None*, *device: Any = device(type='cpu')*, *workspace: pathlib.Path = PosixPath('workspace')*, *compress_dataset: bool = False*, *sampling_patience: int = 500*, ***kwargs: Any*)

Bases: *synthcity.plugins.core.plugin.Plugin*



PATE-GAN: Generating Synthetic Data with Differential Privacy Guarantees.

Parameters

- **generator_n_layers_hidden** – int Number of hidden layers in the generator
- **generator_n_units_hidden** – int Number of hidden units in each layer of the Generator
- **generator_nonlin** – string, default 'leaky_relu' Nonlinearity to use in the generator. Can be 'elu', 'relu', 'selu' or 'leaky_relu'.
- **n_iter** – int Maximum number of iterations in the Generator.
- **generator_dropout** – float Dropout value. If 0, the dropout is not used.
- **discriminator_n_layers_hidden** – int Number of hidden layers in the discriminator
- **discriminator_n_units_hidden** – int Number of hidden units in each layer of the discriminator
- **discriminator_nonlin** – string, default 'leaky_relu' Nonlinearity to use in the discriminator. Can be 'elu', 'relu', 'selu' or 'leaky_relu'.
- **discriminator_n_iter** – int Maximum number of iterations in the discriminator.
- **discriminator_dropout** – float Dropout value for the discriminator. If 0, the dropout is not used.

- **lr** – float learning rate for optimizer.
- **weight_decay** – float l2 (ridge) penalty for the weights.
- **batch_size** – int Batch size
- **random_state** – int random_state used
- **clipping_value** – int, default 0 Gradients clipping value. Zero disables the feature
- **teacher_template** – str Model to use for the teachers. Can be linear, xgboost.
- **epsilon** – float Differential privacy parameter
- **delta** – float Differential privacy parameter
- **lambda** – float Noise size
- **encoder_max_clusters** – int The max number of clusters to create for continuous columns when encoding
- **arguments** (*# Core Plugin*) –
- **workspace** – Path. Optional Path for caching intermediary results.
- **compress_dataset** – bool. Default = False. Drop redundant features before training the generator.
- **sampling_patience** – int. Max inference iterations to wait for the generated data to match the training schema.

Example

```
>>> from sklearn.datasets import load_iris
>>> from synthcity.plugins import Plugins
>>>
>>> X, y = load_iris(as_frame = True, return_X_y = True)
>>> X["target"] = y
>>>
>>> plugin = Plugins().get("pategan", n_iter = 100)
>>> plugin.fit(X)
>>>
>>> plugin.generate(50)
```

class Config

Bases: object

arbitrary_types_allowed = True

validate_assignment = True

fit(*X: Union[synthcity.plugins.core.dataloader.DataLoader, pandas.core.frame.DataFrame]*, **args: Any*, ***kwargs: Any*) → Any

Training method the synthetic data plugin.

Parameters

- **X** – DataLoader. The reference dataset.
- **cond** – Optional, Union[pd.DataFrame, pd.Series, np.ndarray] Optional Training Conditional. The training conditional can be used to control to output of some models, like

GANs or VAEs. The content can be anything, as long as it maps to the training dataset X. Usage example:

```
>>> from sklearn.datasets import load_iris
>>> from synthcity.plugins.core.dataloader import␣
↪GenericDataLoader
>>> from synthcity.plugins.core.constraints import Constraints
>>>
>>> # Load in `test_plugin` the generative model of choice
>>> # ....
>>>
>>> X, y = load_iris(as_frame=True, return_X_y=True)
>>> X["target"] = y
>>>
>>> X = GenericDataLoader(X)
>>> test_plugin.fit(X, cond=y)
>>>
>>> count = 10
>>> X_gen = test_plugin.generate(count, cond=np.ones(count))
>>>
>>> # The Conditional only optimizes the output generation
>>> # for GANs and VAEs, but does NOT guarantee the samples
>>> # are only from that condition.
>>> # If you want to guarantee that output contains only
>>> # "target" == 1 samples, use Constraints.
>>>
>>> constraints = Constraints(
>>>     rules=[
>>>         ("target", "==", 1),
>>>     ]
>>> )
>>> X_gen = test_plugin.generate(count,
>>>         cond=np.ones(count),
>>>         constraints=constraints
>>>        )
>>> assert (X_gen["target"] == 1).all()
```

Returns self

classmethod fqdn() → str
The Fully-Qualified name of the plugin.

generate(*count: Optional[int] = None*, *constraints: Optional[*synthcity.plugins.core.constraints.Constraints*] = None*, *random_state: Optional[int] = None*, ***kwargs: Any*) → *synthcity.plugins.core.dataloader.DataLoader*
Synthetic data generation method.

Parameters

- **count** – optional int. The number of samples to generate. If None, it generated len(reference_dataset) samples.
- **cond** – Optional, Union[pd.DataFrame, pd.Series, np.ndarray]. Optional Generation Conditional. The conditional can be used only if the model was trained using a conditional too. If provided, it must have *count* length. Not all models support conditionals. The conditionals can be used in VAEs or GANs to speed-up the generation under some constraints. For model agnostic solutions, check out the *constraints* parameter.

- **constraints** – optional Constraints. Optional constraints to apply on the generated data. If none, the reference schema constraints are applied. The constraints are model agnostic, and will filter the output of the generative model. The constraints are a list of rules. Each rule is a tuple of the form (<feature>, <operation>, <value>).

  Valid Operations:

  - "<", "lt" : less than <value>
  - "<=", "le": less or equal with <value>
  - ">", "gt" : greater than <value>
  - ">=", "ge": greater or equal with <value>
  - "==", "eq": equal with <value>
  - "in": valid for categorical features, and <value> must be array. for example, ("target", "in", [0, 1])
  - "dtype": <value> can be a data type. For example, ("target", "dtype", "int")

  Usage example:

```
>>> from synthcity.plugins.core.constraints import Constraints
>>> constraints = Constraints(
>>>   rules=[
>>>             ("InterestingFeature", "==", 0),
>>>         ]
>>>     )
>>>
>>> syn_data = syn_model.generate(
        count=count,
        constraints=constraints
    ).dataframe()
>>>
>>> assert (syn_data["InterestingFeature"] == 0).all()
```

- **random_state** – optional int. Optional random seed to use.

Returns <count> synthetic samples

static hyperparameter_space(***kwargs: Any*) → List[*synthcity.plugins.core.distribution.Distribution*]
Returns the hyperparameter space for the derived plugin.

static load(*buff: bytes*) → Any

static load_dict(*representation: dict*) → Any

static name() → str
The name of the plugin.

plot(*plt: Any*, *X:* synthcity.plugins.core.dataloader.DataLoader, *count: Optional[int] = None*, *plots: list = ['marginal', 'associations', 'tsne']*, ***kwargs: Any*) → Any
Plot the real-synthetic distributions.

Parameters

- **plt** – output
- **X** – DataLoader. The reference dataset.

Returns self

classmethod sample_hyperparameters(**args: Any*, ***kwargs: Any*) → Dict[str, Any]
Sample value from the hyperparameter space for the current plugin.

classmethod sample_hyperparameters_optuna(*trial: Any*, **args: Any*, ***kwargs: Any*) → Dict[str, Any]

save() → bytes

save_dict() → dict

save_to_file(*path: pathlib.Path*) → bytes

schema() → *synthcity.plugins.core.schema.Schema*
The reference schema

schema_includes(*other: Union[*synthcity.plugins.core.dataloader.DataLoader, *pandas.core.frame.DataFrame]*) → bool
Helper method to test if the reference schema includes a Dataset

Parameters **other** – DataLoader. The dataset to test

Returns bool, if the schema includes the dataset or not.

training_schema() → *synthcity.plugins.core.schema.Schema*
The internal schema

static type() → str
The type of the plugin.

static version() → str
API version

class Teachers(*n_teachers: int*, *samples_per_teacher: int*, *lamda: float = 0.001*, *template: str = 'xgboost'*)
Bases: *synthcity.plugins.core.serializable.Serializable*

fit(*X: numpy.ndarray*, *generator: Any*) → Any

static load(*buff: bytes*) → Any

static load_dict(*representation: dict*) → Any

pate_lamda(*x: numpy.ndarray*) → Tuple[int, int, int]
Returns PATE_lambda(x).

Parameters **x** (-) – feature vector

Returns the number of label 0 and 1, respectively - out: label after adding laplace noise.

Return type

- n0, n1

save() → bytes

save_dict() → dict

save_to_file(*path: pathlib.Path*) → bytes

static version() → str
API version

plugin
alias of *synthcity.plugins.privacy.plugin_pategan.PATEGANPlugin*

synthcity.plugins.privacy.plugin_privbayes module

Reference: PrivBayes: Private Data Release via Bayesian Networks. (2017), Zhang J, Cormode G, Procopiuc CM, Srivastava D, Xiao X.

class PrivBayes(*epsilon: float = 1.0*, *K: int = 0*, *n_bins: int = 100*, *mi_thresh: float = 0.01*, *target_usefulness: int = 5*)

Bases: *synthcity.plugins.core.serializable.Serializable*

PrivBayes is a differentially private method for releasing high-dimensional data.

Given a dataset D, PrivBayes first constructs a Bayesian network N , which

(i) provides a succinct model of the correlations among the attributes in D

(ii) allows us to approximate the distribution of data in D using a set P of lowdimensional marginals of D.

After that, PrivBayes injects noise into each marginal in P to ensure differential privacy, and then uses the noisy marginals and the Bayesian network to construct an approximation of the data distribution in D. Finally, PrivBayes samples tuples from the approximate distribution to construct a synthetic dataset, and then releases the synthetic data.

display_network() → None

fit(*data: pandas.core.frame.DataFrame*) → Any

static load(*buff: bytes*) → Any

static load_dict(*representation: dict*) → Any

mutual_info_score(*data: pandas.core.frame.DataFrame*, *parents: List[str]*, *candidate: str*) → float
Cluster the source columns, and compute the mutual information between the target and the clusters.

sample(*count: int*) → pandas.core.frame.DataFrame

save() → bytes

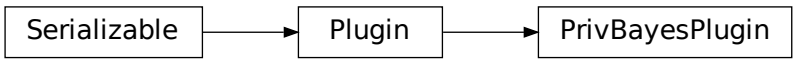
save_dict() → dict

save_to_file(*path: pathlib.Path*) → bytes

static version() → str
API version

class PrivBayesPlugin(*epsilon: float = 1.0*, *K: int = 0*, *n_bins: int = 100*, *mi_thresh: float = 0.01*, *target_usefulness: int = 5*, *random_state: int = 0*, *workspace: pathlib.Path = PosixPath('workspace')*, *compress_dataset: bool = False*, *sampling_patience: int = 500*, ***kwargs: Any*)

Bases: *synthcity.plugins.core.plugin.Plugin*



PrivBayes algorithm.

Args:

epsilon: float Differential privacy parameter

K: Maximum number of parents for a node

n_bins: int Number of bins for encoding the features

mi_thresh: int Mutual information lower threshold. If the current score is lower, the [] parents are used.

target_usefulness: int Def 4.7 in the paper: A noisy distribution is -useful if the ratio of average scale of

information to average scale of noise is no less than . 5-useful is the recommended value.

random_state: int Random seed

Core Plugin arguments workspace: Path.

Optional Path for caching intermediary results.

compress_dataset: bool. Default = False. Drop redundant features before training the generator.

sampling_patience: int. Max inference iterations to wait for the generated data to match the training schema.

Example

```
>>> from sklearn.datasets import load_iris
>>> from synthcity.plugins import Plugins
>>>
>>> X, y = load_iris(as_frame = True, return_X_y = True)
>>> X["target"] = y
>>>
>>> plugin = Plugins().get("privbayes")
>>> plugin.fit(X)
>>>
>>> plugin.generate(50)
```

class Config

Bases: object

arbitrary_types_allowed = True

validate_assignment = True

fit(*X: Union[*synthcity.plugins.core.dataloader.DataLoader*, pandas.core.frame.DataFrame]*, **args: Any*, ***kwargs: Any*) → Any

Training method the synthetic data plugin.

Parameters

- **X** – DataLoader. The reference dataset.
- **cond** – Optional, Union[pd.DataFrame, pd.Series, np.ndarray] Optional Training Conditional. The training conditional can be used to control to output of some models, like GANs or VAEs. The content can be anything, as long as it maps to the training dataset X. Usage example:

```
>>> from sklearn.datasets import load_iris
>>> from synthcity.plugins.core.dataloader import␣
↪GenericDataLoader
>>> from synthcity.plugins.core.constraints import Constraints
>>>
>>> # Load in `test_plugin` the generative model of choice
>>> # ....
>>>
>>> X, y = load_iris(as_frame=True, return_X_y=True)
>>> X["target"] = y
>>>
>>> X = GenericDataLoader(X)
>>> test_plugin.fit(X, cond=y)
>>>
>>> count = 10
>>> X_gen = test_plugin.generate(count, cond=np.ones(count))
>>>
>>> # The Conditional only optimizes the output generation
>>> # for GANs and VAEs, but does NOT guarantee the samples
>>> # are only from that condition.
>>> # If you want to guarantee that output contains only
>>> # "target" == 1 samples, use Constraints.
>>>
>>> constraints = Constraints(
>>>     rules=[
>>>         ("target", "==", 1),
>>>     ]
>>> )
>>> X_gen = test_plugin.generate(count,
>>>         cond=np.ones(count),
>>>         constraints=constraints
>>>        )
>>> assert (X_gen["target"] == 1).all()
```

Returns self

classmethod fqdn() → str
The Fully-Qualified name of the plugin.

generate(*count: Optional[int] = None*, *constraints: Optional[*synthcity.plugins.core.constraints.Constraints*] = None*, *random_state: Optional[int] = None*, ***kwargs: Any*) → *synthcity.plugins.core.dataloader.DataLoader*
Synthetic data generation method.

Parameters

- **count** – optional int. The number of samples to generate. If None, it generated len(reference_dataset) samples.
- **cond** – Optional, Union[pd.DataFrame, pd.Series, np.ndarray]. Optional Generation Conditional. The conditional can be used only if the model was trained using a conditional too. If provided, it must have *count* length. Not all models support conditionals. The conditionals can be used in VAEs or GANs to speed-up the generation under some constraints. For model agnostic solutions, check out the *constraints* parameter.
- **constraints** – optional Constraints. Optional constraints to apply on the generated

data. If none, the reference schema constraints are applied. The constraints are model agnostic, and will filter the output of the generative model. The constraints are a list of rules. Each rule is a tuple of the form (<feature>, <operation>, <value>).

Valid Operations:

- "<", "lt" : less than <value>
- "<=", "le": less or equal with <value>
- ">", "gt" : greater than <value>
- ">=", "ge": greater or equal with <value>
- "==", "eq": equal with <value>
- "in": valid for categorical features, and <value> must be array. for example, ("target", "in", [0, 1])
- "dtype": <value> can be a data type. For example, ("target", "dtype", "int")

Usage example:

```
>>> from synthcity.plugins.core.constraints import Constraints
>>> constraints = Constraints(
>>>   rules=[
>>>             ("InterestingFeature", "==", 0),
>>>         ]
>>>     )
>>>
>>> syn_data = syn_model.generate(
        count=count,
        constraints=constraints
    ).dataframe()
>>>
>>> assert (syn_data["InterestingFeature"] == 0).all()
```

- **random_state** – optional int. Optional random seed to use.

Returns <count> synthetic samples

static hyperparameter_space(***kwargs: Any*) → List[*synthcity.plugins.core.distribution.Distribution*]
Returns the hyperparameter space for the derived plugin.

static load(*buff: bytes*) → Any

static load_dict(*representation: dict*) → Any

static name() → str
The name of the plugin.

plot(*plt: Any*, *X:* synthcity.plugins.core.dataloader.DataLoader, *count: Optional[int] = None*, *plots: list = ['marginal', 'associations', 'tsne']*, ***kwargs: Any*) → Any
Plot the real-synthetic distributions.

Parameters

- **plt** – output
- **X** – DataLoader. The reference dataset.

Returns self

classmethod sample_hyperparameters(**args: Any*, ***kwargs: Any*) → Dict[str, Any]
Sample value from the hyperparameter space for the current plugin.

classmethod sample_hyperparameters_optuna(*trial: Any*, **args: Any*, ***kwargs: Any*) → Dict[str, Any]

save() → bytes

save_dict() → dict

save_to_file(*path: pathlib.Path*) → bytes

schema() → *synthcity.plugins.core.schema.Schema*
The reference schema

schema_includes(*other: Union[*synthcity.plugins.core.dataloader.DataLoader, *pandas.core.frame.DataFrame]*) → bool
Helper method to test if the reference schema includes a Dataset

Parameters **other** – DataLoader. The dataset to test

Returns bool, if the schema includes the dataset or not.

training_schema() → *synthcity.plugins.core.schema.Schema*
The internal schema

static type() → str
The type of the plugin.

static version() → str
API version

class network_edge(*feature*, *parents*)
Bases: tuple

count(*value*, */*)
Return number of occurrences of value.

feature
Alias for field number 0

index(*value*, *start=0*, *stop=9223372036854775807*, */*)
Return first index of value.

Raises ValueError if the value is not present.

parents
Alias for field number 1

plugin
alias of *synthcity.plugins.privacy.plugin_privbayes.PrivBayesPlugin*

usefulness_minus_target(*k: int*, *num_attributes: int*, *num_tuples: int*, *target_usefulness: int = 5*, *epsilon: float = 0.1*) → int
Usefulness function in PrivBayes.

Parameters

- **k** (*int*) – Max number of degree in Bayesian networks construction
- **num_attributes** (*int*) – Number of attributes in dataset.
- **num_tuples** (*int*) – Number of tuples in dataset.
- **target_usefulness** (*int or float*) –

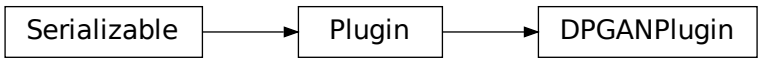
- **epsilon** (*float*) – Parameter of differential privacy.

synthcity.plugins.privacy.plugin_dpgan module

Reference: "Differentially Private Generative Adversarial Network", Xie, Liyang et al.

class DPGANPlugin(*n_iter: int = 2000*, *generator_n_layers_hidden: int = 2*, *generator_n_units_hidden: int = 500*, *generator_nonlin: str = 'relu'*, *generator_dropout: float = 0.1*, *generator_opt_betas: tuple = (0.5, 0.999)*, *discriminator_n_layers_hidden: int = 2*, *discriminator_n_units_hidden: int = 500*, *discriminator_nonlin: str = 'leaky_relu'*, *discriminator_n_iter: int = 1*, *discriminator_dropout: float = 0.1*, *discriminator_opt_betas: tuple = (0.5, 0.999)*, *lr: float = 0.001*, *weight_decay: float = 0.001*, *batch_size: int = 200*, *random_state: int = 0*, *clipping_value: int = 1*, *lambda_gradient_penalty: float = 10*, *encoder_max_clusters: int = 5*, *encoder: Any = None*, *dataloader_sampler: Optional[torch.utils.data.sampler.Sampler] = None*, *device: Any = device(type='cpu')*, *epsilon: float = 1*, *delta: Optional[float] = None*, *dp_max_grad_norm: float = 2*, *dp_secure_mode: bool = False*, *patience: int = 5*, *patience_metric: Optional[synthcity.metrics.weighted_metrics.WeightedMetrics] = None*, *n_iter_print: int = 50*, *n_iter_min: int = 100*, *workspace: pathlib.Path = PosixPath('workspace')*, *compress_dataset: bool = False*, *sampling_patience: int = 500*, ***kwargs: Any*)

Bases: *synthcity.plugins.core.plugin.Plugin*



Differentially Private Generative Adversarial Network implementation. The discriminator is trained using DP-SGD.

Parameters

- **generator_n_layers_hidden** – int Number of hidden layers in the generator
- **generator_n_units_hidden** – int Number of hidden units in each layer of the Generator
- **generator_nonlin** – string, default 'leaky_relu' Nonlinearity to use in the generator. Can be 'elu', 'relu', 'selu' or 'leaky_relu'.
- **n_iter** – int Maximum number of iterations in the Generator.
- **generator_dropout** – float Dropout value. If 0, the dropout is not used.
- **discriminator_n_layers_hidden** – int Number of hidden layers in the discriminator
- **discriminator_n_units_hidden** – int Number of hidden units in each layer of the discriminator
- **discriminator_nonlin** – string, default 'leaky_relu' Nonlinearity to use in the discriminator. Can be 'elu', 'relu', 'selu' or 'leaky_relu'.
- **discriminator_n_iter** – int Maximum number of iterations in the discriminator.
- **discriminator_dropout** – float Dropout value for the discriminator. If 0, the dropout is not used.

- **lr** – float learning rate for optimizer.
- **weight_decay** – float l2 (ridge) penalty for the weights.
- **batch_size** – int Batch size
- **random_state** – int random seed to use
- **clipping_value** – int, default 0 Gradients clipping value. Zero disables the feature
- **encoder_max_clusters** – int The max number of clusters to create for continuous columns when encoding
- **stopping** (*# early*) –
- **n_iter_print** – int Number of iterations after which to print updates and check the validation loss.
- **n_iter_min** – int Minimum number of iterations to go through before starting early stopping
- **patience** – int Max number of iterations without any improvement before early stopping is trigged.
- **patience_metric** – Optional[WeightedMetrics] If not None, the metric is used for evaluation the criterion for early stopping.
- **settings** (*# privacy*) –
- **dp_enabled** – bool Train the discriminator with Differential Privacy guarantees
- **dp_delta** – Optional[float] Optional DP delta: the probability of information accidentally being leaked. Usually 1 / len(dataset)
- **dp_epsilon** – float = 3 DP epsilon: privacy budget, which is a measure of the amount of privacy that is preserved by a given algorithm. Epsilon is a number that represents the maximum amount of information that an adversary can learn about an individual from the output of a differentially private algorithm. The smaller the value of epsilon, the more private the algorithm is. For example, an algorithm with an epsilon of 0.1 preserves more privacy than an algorithm with an epsilon of 1.0.
- **dp_max_grad_norm** – float max grad norm used for gradient clipping
- **dp_secure_mode** – bool = False, if True uses noise generation approach robust to floating point arithmetic attacks.

Example

```
>>> from sklearn.datasets import load_iris
>>> from synthcity.plugins import Plugins
>>>
>>> X, y = load_iris(as_frame = True, return_X_y = True)
>>> X["target"] = y
>>>
>>> plugin = Plugins().get("dpgan", n_iter = 100)
>>> plugin.fit(X)
>>>
>>> plugin.generate(50)
```

Note: There is a known issue with the training step for training GANs with conditionals with dp_enabled set to True, as is the case for DPGAN.

class Config
Bases: object

arbitrary_types_allowed = True

validate_assignment = True

fit(*X: Union[synthcity.plugins.core.dataloader.DataLoader, pandas.core.frame.DataFrame]*, **args: Any*, ***kwargs: Any*) → Any
Training method the synthetic data plugin.

Parameters

- **X** – DataLoader. The reference dataset.
- **cond** – Optional, Union[pd.DataFrame, pd.Series, np.ndarray] Optional Training Conditional. The training conditional can be used to control to output of some models, like GANs or VAEs. The content can be anything, as long as it maps to the training dataset X. Usage example:

```
>>> from sklearn.datasets import load_iris
>>> from synthcity.plugins.core.dataloader import␣
↪GenericDataLoader
>>> from synthcity.plugins.core.constraints import Constraints
>>>
>>> # Load in `test_plugin` the generative model of choice
>>> # ....
>>>
>>> X, y = load_iris(as_frame=True, return_X_y=True)
>>> X["target"] = y
>>>
>>> X = GenericDataLoader(X)
>>> test_plugin.fit(X, cond=y)
>>>
>>> count = 10
>>> X_gen = test_plugin.generate(count, cond=np.ones(count))
>>>
>>> # The Conditional only optimizes the output generation
>>> # for GANs and VAEs, but does NOT guarantee the samples
>>> # are only from that condition.
>>> # If you want to guarantee that output contains only
>>> # "target" == 1 samples, use Constraints.
>>>
>>> constraints = Constraints(
>>>     rules=[
>>>         ("target", "==", 1),
>>>     ]
>>> )
>>> X_gen = test_plugin.generate(count,
>>>         cond=np.ones(count),
>>>         constraints=constraints
>>>        )
>>> assert (X_gen["target"] == 1).all()
```

Returns self

classmethod fqdn() → str
The Fully-Qualified name of the plugin.

generate(*count: Optional[int] = None*, *constraints: Optional[synthcity.plugins.core.constraints.Constraints] = None*, *random_state: Optional[int] = None*, ***kwargs: Any*) → *synthcity.plugins.core.dataloader.DataLoader*

Synthetic data generation method.

Parameters

- **count** – optional int. The number of samples to generate. If None, it generated len(reference_dataset) samples.
- **cond** – Optional, Union[pd.DataFrame, pd.Series, np.ndarray]. Optional Generation Conditional. The conditional can be used only if the model was trained using a conditional too. If provided, it must have *count* length. Not all models support conditionals. The conditionals can be used in VAEs or GANs to speed-up the generation under some constraints. For model agnostic solutions, check out the *constraints* parameter.
- **constraints** – optional Constraints. Optional constraints to apply on the generated data. If none, the reference schema constraints are applied. The constraints are model agnostic, and will filter the output of the generative model. The constraints are a list of rules. Each rule is a tuple of the form (<feature>, <operation>, <value>).

  Valid Operations:

  - "<", "lt" : less than <value>
  - "<=", "le": less or equal with <value>
  - ">", "gt" : greater than <value>
  - ">=", "ge": greater or equal with <value>
  - "==", "eq": equal with <value>
  - "in": valid for categorical features, and <value> must be array. for example, ("target", "in", [0, 1])
  - "dtype": <value> can be a data type. For example, ("target", "dtype", "int")

  Usage example:

```
>>> from synthcity.plugins.core.constraints import Constraints
>>> constraints = Constraints(
>>>   rules=[
>>>             ("InterestingFeature", "==", 0),
>>>         ]
>>>     )
>>>
>>> syn_data = syn_model.generate(
        count=count,
        constraints=constraints
    ).dataframe()
>>>
>>> assert (syn_data["InterestingFeature"] == 0).all()
```

- **random_state** – optional int. Optional random seed to use.

Returns <count> synthetic samples

static hyperparameter_space(***kwargs: Any*) → List[*synthcity.plugins.core.distribution.Distribution*]

Returns the hyperparameter space for the derived plugin.

static load(*buff: bytes*) → Any

static load_dict(*representation: dict*) → Any

static name() → str
The name of the plugin.

plot(*plt: Any*, *X:* synthcity.plugins.core.dataloader.DataLoader, *count: Optional[int] = None*, *plots: list = ['marginal', 'associations', 'tsne']*, ***kwargs: Any*) → Any
Plot the real-synthetic distributions.

Parameters

- **plt** – output
- **X** – DataLoader. The reference dataset.

Returns self

classmethod sample_hyperparameters(**args: Any*, ***kwargs: Any*) → Dict[str, Any]
Sample value from the hyperparameter space for the current plugin.

classmethod sample_hyperparameters_optuna(*trial: Any*, **args: Any*, ***kwargs: Any*) → Dict[str, Any]

save() → bytes

save_dict() → dict

save_to_file(*path: pathlib.Path*) → bytes

schema() → *synthcity.plugins.core.schema.Schema*
The reference schema

schema_includes(*other: Union[*synthcity.plugins.core.dataloader.DataLoader*, pandas.core.frame.DataFrame]*) → bool
Helper method to test if the reference schema includes a Dataset

Parameters **other** – DataLoader. The dataset to test

Returns bool, if the schema includes the dataset or not.

training_schema() → *synthcity.plugins.core.schema.Schema*
The internal schema

static type() → str
The type of the plugin.

static version() → str
API version

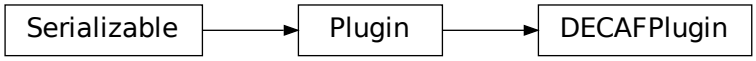
plugin
alias of *synthcity.plugins.privacy.plugin_dpgan.DPGANPlugin*

synthcity.plugins.privacy.plugin_decaf module

Reference: Boris van Breugel, Trent Kyono, Jeroen Berrevoets, Mihaela van der Schaar "DECAF: Generating Fair Synthetic Data Using Causally-Aware Generative Networks"(2021).

class DECAFPlugin(*n_iter: int = 100*, *n_iter_baseline: int = 1000*, *generator_n_layers_hidden: int = 2*, *generator_n_units_hidden: int = 500*, *generator_nonlin: str = 'relu'*, *generator_dropout: float = 0.1*, *generator_opt_betas: tuple = (0.5, 0.999)*, *discriminator_n_layers_hidden: int = 2*, *discriminator_n_units_hidden: int = 500*, *discriminator_nonlin: str = 'leaky_relu'*, *discriminator_n_iter: int = 1*, *discriminator_dropout: float = 0.1*, *discriminator_opt_betas: tuple = (0.5, 0.999)*, *lr: float = 0.001*, *batch_size: int = 200*, *random_state: int = 0*, *clipping_value: int = 1*, *lambda_gradient_penalty: float = 10*, *lambda_privacy: float = 1*, *eps: float = 1e-08*, *alpha: float = 1*, *rho: float = 1*, *weight_decay: float = 0.01*, *l1_g: float = 0*, *l1_W: float = 1*, *grad_dag_loss: bool = False*, *struct_learning_enabled: bool = True*, *struct_learning_n_iter: int = 1000*, *struct_learning_search_method: str = 'tree_search'*, *struct_learning_score: str = 'k2'*, *struct_max_indegree: int = 4*, *encoder_max_clusters: int = 10*, *device: Any = device(type='cpu')*, *workspace: pathlib.Path = PosixPath('workspace')*, *compress_dataset: bool = False*, *sampling_patience: int = 500*, ***kwargs: Any*)

Bases: synthcity.plugins.core.plugin.Plugin



DECAF (DEbiasing CAusal Fairness) plugin.

Parameters

- **n_iter** – int Number of training iterations.
- **generator_n_layers_hidden** – int Number of hidden layers in the generator.
- **generator_n_units_hidden** – Number of neurons in the hidden layers of the generator.
- **generator_nonlin** – str Nonlinearity used by the generator for the hidden layers: leaky_relu, relu, gelu etc.
- **generator_dropout** – float Generator dropout.
- **generator_opt_betas** – tuple Generator initial decay rates for the Adam optimizer
- **discriminator_n_layers_hidden** – int Number of hidden layers in the discriminator.
- **discriminator_n_units_hidden** – int Number of neurons in the hidden layers of the discriminator.
- **discriminator_nonlin** – str Nonlinearity used by the discriminator for the hidden layers: leaky_relu, relu, gelu etc.
- **discriminator_n_iter** – int Discriminator number of iterations(default = 1)
- **discriminator_dropout** – float Discriminator dropout
- **discriminator_opt_betas** – tuple Discriminator initial decay rates for the Adam optimizer
- **lr** – float Learning rate
- **weight_decay** – float Optimizer weight decay
- **batch_size** – int Batch size
- **random_state** – int Random seed

- **clipping_value** – int Gradient clipping value
- **lambda_gradient_penalty** – float Gradient penalty factor used for training the GAN.
- **lambda_privacy** – float Privacy factor used the AdsGAN loss.
- **eps** – float = 1e-8, Noise added to the privacy loss
- **alpha** – float Gradient penalty weight for real samples.
- **rho** – float DAG loss factor
- **l1_g** – float = 0 l1 regularization loss for the generator
- **l1_W** – float = 1 l1 regularization factor for l1_g
- **struct_learning_enabled** – bool Enable DAG learning outside DECAF.
- **struct_learning_n_iter** – int Number of iterations for the DAG search.
- **struct_learning_search_method** – str DAG search strategy: hillclimb, pc, tree_search, mmhc, exhaustive, d-struct
- **struct_learning_score** – str DAG search scoring strategy: k2, bdeu, bic, bds
- **struct_max_indegree** – int Max parents in the DAG.
- **encoder_max_clusters** – int Number of clusters used for tabular encoding
- **device** – Any = DEVICE torch device used for training.
- **arguments** (*# Core Plugin*) –
- **workspace** – Path. Optional Path for caching intermediary results.
- **compress_dataset** – bool. Default = False. Drop redundant features before training the generator.
- **sampling_patience** – int. Max inference iterations to wait for the generated data to match the training schema.

Example

```
>>> from sklearn.datasets import load_iris
>>> from synthcity.plugins import Plugins
>>>
>>> X, y = load_iris(as_frame = True, return_X_y = True)
>>> X["target"] = y
>>>
>>> plugin = Plugins().get("decaf", n_iter = 100)
>>> plugin.fit(X)
>>>
>>> plugin.generate(50)
```

class Config

Bases: object

arbitrary_types_allowed = True

validate_assignment = True

fit(*X: Union[synthcity.plugins.core.dataloader.DataLoader, pandas.core.frame.DataFrame]*, **args: Any*, ***kwargs: Any*) → Any
Training method the synthetic data plugin.

Parameters

- **X** – DataLoader. The reference dataset.
- **cond** – Optional, Union[pd.DataFrame, pd.Series, np.ndarray] Optional Training Conditional. The training conditional can be used to control to output of some models, like GANs or VAEs. The content can be anything, as long as it maps to the training dataset X. Usage example:

```
>>> from sklearn.datasets import load_iris
>>> from synthcity.plugins.core.dataloader import␣
↪GenericDataLoader
>>> from synthcity.plugins.core.constraints import Constraints
>>>
>>> # Load in `test_plugin` the generative model of choice
>>> # ....
>>>
>>> X, y = load_iris(as_frame=True, return_X_y=True)
>>> X["target"] = y
>>>
>>> X = GenericDataLoader(X)
>>> test_plugin.fit(X, cond=y)
>>>
>>> count = 10
>>> X_gen = test_plugin.generate(count, cond=np.ones(count))
>>>
>>> # The Conditional only optimizes the output generation
>>> # for GANs and VAEs, but does NOT guarantee the samples
>>> # are only from that condition.
>>> # If you want to guarantee that output contains only
>>> # "target" == 1 samples, use Constraints.
>>>
>>> constraints = Constraints(
>>>     rules=[
>>>         ("target", "==", 1),
>>>     ]
>>> )
>>> X_gen = test_plugin.generate(count,
>>>         cond=np.ones(count),
>>>         constraints=constraints
>>>        )
>>> assert (X_gen["target"] == 1).all()
```

Returns self

classmethod fqdn() → str
The Fully-Qualified name of the plugin.

generate(*count: Optional[int] = None*, *constraints: Optional[synthcity.plugins.core.constraints.Constraints] = None*, *random_state: Optional[int] = None*, ***kwargs: Any*) → *synthcity.plugins.core.dataloader.DataLoader*
Synthetic data generation method.

Parameters

- **count** – optional int. The number of samples to generate. If None, it generated len(reference_dataset) samples.
- **cond** – Optional, Union[pd.DataFrame, pd.Series, np.ndarray]. Optional Generation Conditional. The conditional can be used only if the model was trained using a conditional too. If provided, it must have *count* length. Not all models support conditionals. The conditionals can be used in VAEs or GANs to speed-up the generation under some constraints. For model agnostic solutions, check out the *constraints* parameter.
- **constraints** – optional Constraints. Optional constraints to apply on the generated data. If none, the reference schema constraints are applied. The constraints are model agnostic, and will filter the output of the generative model. The constraints are a list of rules. Each rule is a tuple of the form (<feature>, <operation>, <value>).

  Valid Operations:

  - "<", "lt" : less than <value>
  - "<=", "le": less or equal with <value>
  - ">", "gt" : greater than <value>
  - ">=", "ge": greater or equal with <value>
  - "==", "eq": equal with <value>
  - "in": valid for categorical features, and <value> must be array. for example, ("target", "in", [0, 1])
  - "dtype": <value> can be a data type. For example, ("target", "dtype", "int")

  Usage example:

```
>>> from synthcity.plugins.core.constraints import Constraints
>>> constraints = Constraints(
>>>   rules=[
>>>             ("InterestingFeature", "==", 0),
>>>         ]
>>>     )
>>>
>>> syn_data = syn_model.generate(
        count=count,
        constraints=constraints
    ).dataframe()
>>>
>>> assert (syn_data["InterestingFeature"] == 0).all()
```

- **random_state** – optional int. Optional random seed to use.

Returns <count> synthetic samples

get_dag(*X: pandas.core.frame.DataFrame*, *struct_learning_search_method: Optional[str] = None*, *as_index: bool = False*) → Any

static hyperparameter_space(***kwargs: Any*) → List[*synthcity.plugins.core.distribution.Distribution*]

Returns the hyperparameter space for the derived plugin.

static load(*buff: bytes*) → Any

static load_dict(*representation: dict*) → Any

static name() → str
The name of the plugin.

plot(*plt: Any*, *X:* synthcity.plugins.core.dataloader.DataLoader, *count: Optional[int] = None*, *plots: list = ['marginal', 'associations', 'tsne']*, ***kwargs: Any*) → Any
Plot the real-synthetic distributions.

Parameters

- **plt** – output
- **X** – DataLoader. The reference dataset.

Returns self

classmethod sample_hyperparameters(**args: Any*, ***kwargs: Any*) → Dict[str, Any]
Sample value from the hyperparameter space for the current plugin.

classmethod sample_hyperparameters_optuna(*trial: Any*, **args: Any*, ***kwargs: Any*) → Dict[str, Any]

save() → bytes

save_dict() → dict

save_to_file(*path: pathlib.Path*) → bytes

schema() → *synthcity.plugins.core.schema.Schema*
The reference schema

schema_includes(*other: Union[*synthcity.plugins.core.dataloader.DataLoader*, pandas.core.frame.DataFrame]*) → bool
Helper method to test if the reference schema includes a Dataset

Parameters **other** – DataLoader. The dataset to test

Returns bool, if the schema includes the dataset or not.

training_schema() → *synthcity.plugins.core.schema.Schema*
The internal schema

static type() → str
The type of the plugin.

static version() → str
API version

plugin
alias of *synthcity.plugins.privacy.plugin_decaf.DECAFPlugin*

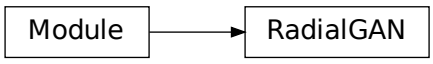
10.1.3 Domain adaptation

synthcity.plugins.domain_adaptation.plugin_radialgan module

Reference: Yoon, Jinsung and Jordon, James and van der Schaar, Mihaela "RadialGAN: Leveraging multiple datasets to improve target-specific predictive models using Generative Adversarial Networks"

class RadialGAN(*domains: List[int]*, *n_features: int*, *n_units_latent: int*, *generator_n_layers_hidden: int = 2*, *generator_n_units_hidden: int = 250*, *generator_nonlin: str = 'leaky_relu'*, *generator_nonlin_out: Optional[List[Tuple[str, int]]] = None*, *generator_n_iter: int = 500*, *generator_dropout: float = 0*, *generator_lr: float = 0.0002*, *generator_weight_decay: float = 0.001*, *generator_opt_betas: tuple = (0.9, 0.999)*, *discriminator_n_layers_hidden: int = 3*, *discriminator_n_units_hidden: int = 300*, *discriminator_nonlin: str = 'leaky_relu'*, *discriminator_n_iter: int = 1*, *discriminator_dropout: float = 0.1*, *discriminator_lr: float = 0.0002*, *discriminator_weight_decay: float = 0.001*, *discriminator_opt_betas: tuple = (0.9, 0.999)*, *batch_size: int = 64*, *n_iter_print: int = 10*, *random_state: int = 0*, *clipping_value: int = 0*, *lambda_gradient_penalty: float = 10*, *device: Any = device(type='cpu')*, *dataloader_sampler: Any = None*)

Bases: `torch.nn.modules.module.Module`



RadialGAN implementation: Leveraging multiple datasets to improve target-specific predictive models using Generative Adversarial Networks.

Parameters

- **domains** – List[int] List of domains
- **n_features** – int Number of features
- **n_units_latent** – int Number of hidden units
- **generator_n_layers_hidden** – int Number of hidden layers in the generator
- **generator_n_units_hidden** – int Number of hidden units in each layer of the Generator
- **generator_nonlin** – string, default 'elu' Nonlinearity to use in the generator. Can be 'elu', 'relu', 'selu' or 'leaky_relu'.
- **generator_n_iter** – int Maximum number of iterations in the Generator.
- **generator_dropout** – float Dropout value. If 0, the dropout is not used.
- **discriminator_n_layers_hidden** – int Number of hidden layers in the discriminator
- **discriminator_n_units_hidden** – int Number of hidden units in each layer of the discriminator
- **discriminator_nonlin** – string, default 'relu' Nonlinearity to use in the discriminator. Can be 'elu', 'relu', 'selu' or 'leaky_relu'.
- **discriminator_n_iter** – int Maximum number of iterations in the discriminator.
- **discriminator_dropout** – float Dropout value for the discriminator. If 0, the dropout is not used.
- **lr** – float learning rate for optimizer.
- **weight_decay** – float l2 (ridge) penalty for the weights.
- **batch_size** – int Batch size

- **n_iter_print** – int Number of iterations after which to print updates and check the validation loss.
- **random_state** – int random_state used
- **clipping_value** – int, default 0 Gradients clipping value. Zero disables the feature

T_destination
alias of TypeVar('T_destination', bound=Dict[str, Any])

add_module(*name: str*, *module: Optional[torch.nn.modules.module.Module]*) → None
Adds a child module to the current module.

The module can be accessed as an attribute using the given name.

Parameters

- **name** (*str*) – name of the child module. The child module can be accessed from this module using the given name
- **module** (*Module*) – child module to be added to the module.

apply(*fn: Callable[[torch.nn.modules.module.Module], None]*) → torch.nn.modules.module.T
Applies `fn` recursively to every submodule (as returned by `.children()`) as well as self. Typical use includes initializing the parameters of a model (see also nn-init-doc).

Parameters **fn** (Module -> None) – function to be applied to each submodule

Returns self

Return type Module

Example:

```
>>> @torch.no_grad()
>>> def init_weights(m):
>>>     print(m)
>>>     if type(m) == nn.Linear:
>>>         m.weight.fill_(1.0)
>>>         print(m.weight)
>>> net = nn.Sequential(nn.Linear(2, 2), nn.Linear(2, 2))
>>> net.apply(init_weights)
Linear(in_features=2, out_features=2, bias=True)
Parameter containing:
tensor([[1., 1.],
        [1., 1.]], requires_grad=True)
Linear(in_features=2, out_features=2, bias=True)
Parameter containing:
tensor([[1., 1.],
        [1., 1.]], requires_grad=True)
Sequential(
  (0): Linear(in_features=2, out_features=2, bias=True)
  (1): Linear(in_features=2, out_features=2, bias=True)
)
```

bfloat16() → torch.nn.modules.module.T
Casts all floating point parameters and buffers to `bfloat16` datatype.

Note: This method modifies the module in-place.

Returns self

Return type Module

buffers(*recurse: bool = True*) → Iterator[torch.Tensor]
Returns an iterator over module buffers.

Parameters **recurse** (*bool*) – if True, then yields buffers of this module and all submodules. Otherwise, yields only buffers that are direct members of this module.

Yields *torch.Tensor* – module buffer

Example:

```
>>> # xdoctest: +SKIP("undefined vars")
>>> for buf in model.buffers():
>>>     print(type(buf), buf.size())
<class 'torch.Tensor'> (20L,)
<class 'torch.Tensor'> (20L, 1L, 5L, 5L)
```

children() → Iterator[torch.nn.modules.module.Module]
Returns an iterator over immediate children modules.

Yields *Module* – a child module

cpu() → torch.nn.modules.module.T
Moves all model parameters and buffers to the CPU.

Note: This method modifies the module in-place.

Returns self

Return type Module

cuda(*device: Optional[Union[int, torch.device]] = None*) → torch.nn.modules.module.T
Moves all model parameters and buffers to the GPU.

This also makes associated parameters and buffers different objects. So it should be called before constructing optimizer if the module will live on GPU while being optimized.

Note: This method modifies the module in-place.

Parameters **device** (*int, optional*) – if specified, all parameters will be copied to that device

Returns self

Return type Module

dataloader(*X: torch.Tensor*, *domains: torch.Tensor*) → torch.utils.data.dataloader.DataLoader

double() → torch.nn.modules.module.T
Casts all floating point parameters and buffers to `double` datatype.

Note: This method modifies the module in-place.

Returns self

Return type Module

dump_patches: bool = False

eval() → torch.nn.modules.module.T
Sets the module in evaluation mode.

This has any effect only on certain modules. See documentations of particular modules for details of their behaviors in training/evaluation mode, if they are affected, e.g. `Dropout`, `BatchNorm`, etc.

This is equivalent with `self.train(False)`.

See locally-disable-grad-doc for a comparison between *.eval()* and several similar mechanisms that may be confused with it.

Returns self

Return type Module

extra_repr() → str
Set the extra representation of the module

To print customized extra information, you should re-implement this method in your own modules. Both single-line and multi-line strings are acceptable.

fit(*X: numpy.ndarray*, *domains: numpy.ndarray*) →
synthcity.plugins.domain_adaptation.plugin_radialgan.RadialGAN

float() → torch.nn.modules.module.T
Casts all floating point parameters and buffers to `float` datatype.

Note: This method modifies the module in-place.

Returns self

Return type Module

forward(*count: int*, *domains: Optional[List[int]] = None*) → Tuple[torch.Tensor, List]
Defines the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

---

generate(*count: int*, *domains: Optional[List[int]] = None*) → Tuple[numpy.ndarray, numpy.ndarray]

get_buffer(*target: str*) → torch.Tensor
Returns the buffer given by `target` if it exists, otherwise throws an error.

See the docstring for `get_submodule` for a more detailed explanation of this method's functionality as well as how to correctly specify `target`.

Parameters **target** – The fully-qualified string name of the buffer to look for. (See `get_submodule` for how to specify a fully-qualified string.)

Returns The buffer referenced by `target`

Return type torch.Tensor

Raises **AttributeError** – If the target string references an invalid path or resolves to something that is not a buffer

get_extra_state() → Any

Returns any extra state to include in the module's state_dict. Implement this and a corresponding *set_extra_state()* for your module if you need to store extra state. This function is called when building the module's *state_dict()*.

Note that extra state should be pickleable to ensure working serialization of the state_dict. We only provide provide backwards compatibility guarantees for serializing Tensors; other objects may break backwards compatibility if their serialized pickled form changes.

Returns Any extra state to store in the module's state_dict

Return type object

get_parameter(*target: str*) → torch.nn.parameter.Parameter

Returns the parameter given by `target` if it exists, otherwise throws an error.

See the docstring for `get_submodule` for a more detailed explanation of this method's functionality as well as how to correctly specify `target`.

Parameters **target** – The fully-qualified string name of the Parameter to look for. (See `get_submodule` for how to specify a fully-qualified string.)

Returns The Parameter referenced by `target`

Return type torch.nn.Parameter

Raises **AttributeError** – If the target string references an invalid path or resolves to something that is not an `nn.Parameter`

get_submodule(*target: str*) → torch.nn.modules.module.Module

Returns the submodule given by `target` if it exists, otherwise throws an error.

For example, let's say you have an `nn.Module` `A` that looks like this:

```
A(
    (net_b): Module(
        (net_c): Module(
            (conv): Conv2d(16, 33, kernel_size=(3, 3), stride=(2, 2))
        )
        (linear): Linear(in_features=100, out_features=200, bias=True)
    )
)
```

(The diagram shows an `nn.Module` `A`. `A` has a nested submodule `net_b`, which itself has two submodules `net_c` and `linear`. `net_c` then has a submodule `conv`.)

To check whether or not we have the `linear` submodule, we would call `get_submodule("net_b.linear")`. To check whether we have the `conv` submodule, we would call `get_submodule("net_b.net_c.conv")`.

The runtime of `get_submodule` is bounded by the degree of module nesting in `target`. A query against `named_modules` achieves the same result, but it is O(N) in the number of transitive modules. So, for a simple check to see if some submodule exists, `get_submodule` should always be used.

Parameters **target** – The fully-qualified string name of the submodule to look for. (See above example for how to specify a fully-qualified string.)

Returns The submodule referenced by `target`

Return type torch.nn.Module

Raises **AttributeError** – If the target string references an invalid path or resolves to something that is not an nn.Module

half() → torch.nn.modules.module.T
Casts all floating point parameters and buffers to half datatype.

Note: This method modifies the module in-place.

Returns self

Return type Module

ipu(*device: Optional[Union[int, torch.device]] = None*) → torch.nn.modules.module.T
Moves all model parameters and buffers to the IPU.

This also makes associated parameters and buffers different objects. So it should be called before constructing optimizer if the module will live on IPU while being optimized.

Note: This method modifies the module in-place.

Parameters **device** (*int, optional*) – if specified, all parameters will be copied to that device

Returns self

Return type Module

load_state_dict(*state_dict: Mapping[str, Any]*, *strict: bool = True*)
Copies parameters and buffers from *state_dict* into this module and its descendants. If strict is True, then the keys of *state_dict* must exactly match the keys returned by this module's state_dict() function.

Parameters

- **state_dict** (*dict*) – a dict containing parameters and persistent buffers.
- **strict** (*bool, optional*) – whether to strictly enforce that the keys in *state_dict* match the keys returned by this module's state_dict() function. Default: True

Returns

- **missing_keys** is a list of str containing the missing keys
- **unexpected_keys** is a list of str containing the unexpected keys

Return type NamedTuple with missing_keys and unexpected_keys fields

---

Note: If a parameter or buffer is registered as None and its corresponding key exists in *state_dict*, *load_state_dict()* will raise a RuntimeError.

---

modules() → Iterator[torch.nn.modules.module.Module]
Returns an iterator over all modules in the network.

Yields *Module* – a module in the network

Note: Duplicate modules are returned only once. In the following example, `l` will be returned only once.

Example:

```
>>> l = nn.Linear(2, 2)
>>> net = nn.Sequential(l, l)
>>> for idx, m in enumerate(net.modules()):
...     print(idx, '->', m)

0 -> Sequential(
  (0): Linear(in_features=2, out_features=2, bias=True)
  (1): Linear(in_features=2, out_features=2, bias=True)
)
1 -> Linear(in_features=2, out_features=2, bias=True)
```

named_buffers(*prefix: str = ''*, *recurse: bool = True*) → Iterator[Tuple[str, torch.Tensor]]

Returns an iterator over module buffers, yielding both the name of the buffer as well as the buffer itself.

Parameters

- **prefix** (*str*) – prefix to prepend to all buffer names.
- **recurse** (*bool*) – if True, then yields buffers of this module and all submodules. Otherwise, yields only buffers that are direct members of this module.

Yields *(str, torch.Tensor)* – Tuple containing the name and buffer

Example:

```
>>> # xdoctest: +SKIP("undefined vars")
>>> for name, buf in self.named_buffers():
>>>    if name in ['running_var']:
>>>        print(buf.size())
```

named_children() → Iterator[Tuple[str, torch.nn.modules.module.Module]]

Returns an iterator over immediate children modules, yielding both the name of the module as well as the module itself.

Yields *(str, Module)* – Tuple containing a name and child module

Example:

```
>>> # xdoctest: +SKIP("undefined vars")
>>> for name, module in model.named_children():
>>>     if name in ['conv4', 'conv5']:
>>>         print(module)
```

named_modules(*memo: Optional[Set[torch.nn.modules.module.Module]] = None*, *prefix: str = ''*, *remove_duplicate: bool = True*)

Returns an iterator over all modules in the network, yielding both the name of the module as well as the module itself.

Parameters

- **memo** – a memo to store the set of modules already added to the result
- **prefix** – a prefix that will be added to the name of the module

- **remove_duplicate** – whether to remove the duplicated module instances in the result or not

Yields *(str, Module)* – Tuple of name and module

Note: Duplicate modules are returned only once. In the following example, `l` will be returned only once.

Example:

```
>>> l = nn.Linear(2, 2)
>>> net = nn.Sequential(l, l)
>>> for idx, m in enumerate(net.named_modules()):
...     print(idx, '->', m)

0 -> ('', Sequential(
  (0): Linear(in_features=2, out_features=2, bias=True)
  (1): Linear(in_features=2, out_features=2, bias=True)
))
1 -> ('0', Linear(in_features=2, out_features=2, bias=True))
```

named_parameters(*prefix: str = ''*, *recurse: bool = True*) → Iterator[Tuple[str, torch.nn.parameter.Parameter]]

Returns an iterator over module parameters, yielding both the name of the parameter as well as the parameter itself.

Parameters

- **prefix** (*str*) – prefix to prepend to all parameter names.
- **recurse** (*bool*) – if True, then yields parameters of this module and all submodules. Otherwise, yields only parameters that are direct members of this module.

Yields *(str, Parameter)* – Tuple containing the name and parameter

Example:

```
>>> # xdoctest: +SKIP("undefined vars")
>>> for name, param in self.named_parameters():
>>>    if name in ['bias']:
>>>        print(param.size())
```

parameters(*recurse: bool = True*) → Iterator[torch.nn.parameter.Parameter]

Returns an iterator over module parameters.

This is typically passed to an optimizer.

Parameters **recurse** (*bool*) – if True, then yields parameters of this module and all submodules. Otherwise, yields only parameters that are direct members of this module.

Yields *Parameter* – module parameter

Example:

```
>>> # xdoctest: +SKIP("undefined vars")
>>> for param in model.parameters():
>>>     print(type(param), param.size())
<class 'torch.Tensor'> (20L,)
<class 'torch.Tensor'> (20L, 1L, 5L, 5L)
```

register_backward_hook(*hook: Callable[[torch.nn.modules.module.Module, Union[Tuple[torch.Tensor, ...], torch.Tensor], Union[Tuple[torch.Tensor, ...], torch.Tensor]], Union[None, torch.Tensor]]*) → torch.utils.hooks.RemovableHandle

Registers a backward hook on the module.

This function is deprecated in favor of `register_full_backward_hook()` and the behavior of this function will change in future versions.

Returns a handle that can be used to remove the added hook by calling `handle.remove()`

Return type `torch.utils.hooks.RemovableHandle`

register_buffer(*name: str*, *tensor: Optional[torch.Tensor]*, *persistent: bool = True*) → None

Adds a buffer to the module.

This is typically used to register a buffer that should not to be considered a model parameter. For example, BatchNorm's `running_mean` is not a parameter, but is part of the module's state. Buffers, by default, are persistent and will be saved alongside parameters. This behavior can be changed by setting `persistent` to `False`. The only difference between a persistent buffer and a non-persistent buffer is that the latter will not be a part of this module's *state_dict*.

Buffers can be accessed as attributes using given names.

Parameters

- **name** (*str*) – name of the buffer. The buffer can be accessed from this module using the given name
- **tensor** (*Tensor or None*) – buffer to be registered. If `None`, then operations that run on buffers, such as *cuda*, are ignored. If `None`, the buffer is **not** included in the module's *state_dict*.
- **persistent** (*bool*) – whether the buffer is part of this module's *state_dict*.

Example:

```
>>> # xdoctest: +SKIP("undefined vars")
>>> self.register_buffer('running_mean', torch.zeros(num_features))
```

register_forward_hook(*hook: Callable[[...], None]*) → torch.utils.hooks.RemovableHandle

Registers a forward hook on the module.

The hook will be called every time after *forward()* has computed an output. It should have the following signature:

```
hook(module, input, output) -> None or modified output
```

The input contains only the positional arguments given to the module. Keyword arguments won't be passed to the hooks and only to the `forward`. The hook can modify the output. It can modify the input inplace but it will not have effect on forward since this is called after *forward()* is called.

Returns a handle that can be used to remove the added hook by calling `handle.remove()`

Return type `torch.utils.hooks.RemovableHandle`

register_forward_pre_hook(*hook: Callable[[...], None]*) → torch.utils.hooks.RemovableHandle

Registers a forward pre-hook on the module.

The hook will be called every time before *forward()* is invoked. It should have the following signature:

```
hook(module, input) -> None or modified input
```

The input contains only the positional arguments given to the module. Keyword arguments won't be passed to the hooks and only to the `forward`. The hook can modify the input. User can either return a tuple or a single modified value in the hook. We will wrap the value into a tuple if a single value is returned(unless that value is already a tuple).

Returns a handle that can be used to remove the added hook by calling `handle.remove()`

Return type `torch.utils.hooks.RemovableHandle`

register_full_backward_hook(*hook: Callable[[torch.nn.modules.module.Module, Union[Tuple[torch.Tensor, ...], torch.Tensor], Union[Tuple[torch.Tensor, ...], torch.Tensor]], Union[None, torch.Tensor]]*) → torch.utils.hooks.RemovableHandle

Registers a backward hook on the module.

The hook will be called every time the gradients with respect to a module are computed, i.e. the hook will execute if and only if the gradients with respect to module outputs are computed. The hook should have the following signature:

```
hook(module, grad_input, grad_output) -> tuple(Tensor) or None
```

The `grad_input` and `grad_output` are tuples that contain the gradients with respect to the inputs and outputs respectively. The hook should not modify its arguments, but it can optionally return a new gradient with respect to the input that will be used in place of `grad_input` in subsequent computations. `grad_input` will only correspond to the inputs given as positional arguments and all kwarg arguments are ignored. Entries in `grad_input` and `grad_output` will be `None` for all non-Tensor arguments.

For technical reasons, when this hook is applied to a Module, its forward function will receive a view of each Tensor passed to the Module. Similarly the caller will receive a view of each Tensor returned by the Module's forward function.

Warning: Modifying inputs or outputs inplace is not allowed when using backward hooks and will raise an error.

Returns a handle that can be used to remove the added hook by calling `handle.remove()`

Return type `torch.utils.hooks.RemovableHandle`

register_load_state_dict_post_hook(*hook*)

Registers a post hook to be run after module's `load_state_dict` is called.

It should have the following signature:: hook(module, incompatible_keys) -> None

The `module` argument is the current module that this hook is registered on, and the `incompatible_keys` argument is a `NamedTuple` consisting of attributes `missing_keys` and `unexpected_keys`. `missing_keys` is a `list` of `str` containing the missing keys and `unexpected_keys` is a `list` of `str` containing the unexpected keys.

The given incompatible_keys can be modified inplace if needed.

Note that the checks performed when calling *load_state_dict()* with `strict=True` are affected by modifications the hook makes to `missing_keys` or `unexpected_keys`, as expected. Additions to either set of keys will result in an error being thrown when `strict=True`, and clearning out both missing and unexpected keys will avoid an error.

Returns a handle that can be used to remove the added hook by calling `handle.remove()`

Return type `torch.utils.hooks.RemovableHandle`

register_module(*name: str*, *module: Optional[torch.nn.modules.module.Module]*) → None
Alias for *add_module()*.

register_parameter(*name: str*, *param: Optional[torch.nn.parameter.Parameter]*) → None
Adds a parameter to the module.

The parameter can be accessed as an attribute using given name.

Parameters

- **name** (*str*) – name of the parameter. The parameter can be accessed from this module using the given name
- **param** (*Parameter or None*) – parameter to be added to the module. If `None`, then operations that run on parameters, such as *cuda*, are ignored. If `None`, the parameter is **not** included in the module's *state_dict*.

requires_grad_(*requires_grad: bool = True*) → torch.nn.modules.module.T
Change if autograd should record operations on parameters in this module.

This method sets the parameters' `requires_grad` attributes in-place.

This method is helpful for freezing part of the module for finetuning or training parts of a model individually (e.g., GAN training).

See locally-disable-grad-doc for a comparison between *.requires_grad_()* and several similar mechanisms that may be confused with it.

Parameters **requires_grad** (*bool*) – whether autograd should record operations on parameters in this module. Default: `True`.

Returns self

Return type Module

set_extra_state(*state: Any*)
This function is called from *load_state_dict()* to handle any extra state found within the *state_dict*. Implement this function and a corresponding *get_extra_state()* for your module if you need to store extra state within its *state_dict*.

Parameters **state** (*dict*) – Extra state from the *state_dict*

share_memory() → torch.nn.modules.module.T
See `torch.Tensor.share_memory_()`

state_dict(**args*, *destination=None*, *prefix=''*, *keep_vars=False*)
Returns a dictionary containing references to the whole state of the module.

Both parameters and persistent buffers (e.g. running averages) are included. Keys are corresponding parameter and buffer names. Parameters and buffers set to `None` are not included.

Note: The returned object is a shallow copy. It contains references to the module's parameters and buffers.

Warning: Currently `state_dict()` also accepts positional arguments for `destination`, `prefix` and `keep_vars` in order. However, this is being deprecated and keyword arguments will be enforced in future releases.

Warning: Please avoid the use of argument `destination` as it is not designed for end-users.

Parameters

- **destination** (*dict, optional*) – If provided, the state of module will be updated into the dict and the same object is returned. Otherwise, an `OrderedDict` will be created and returned. Default: `None`.
- **prefix** (*str, optional*) – a prefix added to parameter and buffer names to compose the keys in state_dict. Default: `''`.
- **keep_vars** (*bool, optional*) – by default the `Tensor` s returned in the state dict are detached from autograd. If it's set to `True`, detaching will not be performed. Default: `False`.

Returns a dictionary containing a whole state of the module

Return type dict

Example:

```
>>> # xdoctest: +SKIP("undefined vars")
>>> module.state_dict().keys()
['bias', 'weight']
```

to(**args*, ***kwargs*)

Moves and/or casts the parameters and buffers.

This can be called as

to(*device=None*, *dtype=None*, *non_blocking=False*)

to(*dtype*, *non_blocking=False*)

to(*tensor*, *non_blocking=False*)

to(*memory_format=torch.channels_last*)

Its signature is similar to `torch.Tensor.to()`, but only accepts floating point or complex `dtype`s. In addition, this method will only cast the floating point or complex parameters and buffers to `dtype` (if given). The integral parameters and buffers will be moved `device`, if that is given, but with dtypes unchanged. When `non_blocking` is set, it tries to convert/move asynchronously with respect to the host if possible, e.g., moving CPU Tensors with pinned memory to CUDA devices.

See below for examples.

Note: This method modifies the module in-place.

Parameters

- **device** (`torch.device`) – the desired device of the parameters and buffers in this module
- **dtype** (`torch.dtype`) – the desired floating point or complex dtype of the parameters and buffers in this module
- **tensor** (*torch.Tensor*) – Tensor whose dtype and device are the desired dtype and device for all parameters and buffers in this module

- **memory_format** (torch.memory_format) – the desired memory format for 4D parameters and buffers in this module (keyword only argument)

Returns self

Return type Module

Examples:

```
>>> # xdoctest: +IGNORE_WANT("non-deterministic")
>>> linear = nn.Linear(2, 2)
>>> linear.weight
Parameter containing:
tensor([[ 0.1913, -0.3420],
        [-0.5113, -0.2325]])
>>> linear.to(torch.double)
Linear(in_features=2, out_features=2, bias=True)
>>> linear.weight
Parameter containing:
tensor([[ 0.1913, -0.3420],
        [-0.5113, -0.2325]], dtype=torch.float64)
>>> # xdoctest: +REQUIRES(env:TORCH_DOCTEST_CUDA1)
>>> gpu1 = torch.device("cuda:1")
>>> linear.to(gpu1, dtype=torch.half, non_blocking=True)
Linear(in_features=2, out_features=2, bias=True)
>>> linear.weight
Parameter containing:
tensor([[ 0.1914, -0.3420],
        [-0.5112, -0.2324]], dtype=torch.float16, device='cuda:1')
>>> cpu = torch.device("cpu")
>>> linear.to(cpu)
Linear(in_features=2, out_features=2, bias=True)
>>> linear.weight
Parameter containing:
tensor([[ 0.1914, -0.3420],
        [-0.5112, -0.2324]], dtype=torch.float16)

>>> linear = nn.Linear(2, 2, bias=None).to(torch.cdouble)
>>> linear.weight
Parameter containing:
tensor([[ 0.3741+0.j,  0.2382+0.j],
        [ 0.5593+0.j, -0.4443+0.j]], dtype=torch.complex128)
>>> linear(torch.ones(3, 2, dtype=torch.cdouble))
tensor([[0.6122+0.j, 0.1150+0.j],
        [0.6122+0.j, 0.1150+0.j],
        [0.6122+0.j, 0.1150+0.j]], dtype=torch.complex128)
```

to_empty(*, *device: Union[str, torch.device]*) → torch.nn.modules.module.T

Moves the parameters and buffers to the specified device without copying storage.

Parameters **device** (torch.device) – The desired device of the parameters and buffers in this module.

Returns self

Return type Module

train(*mode: bool = True*) → torch.nn.modules.module.T
Sets the module in training mode.

This has any effect only on certain modules. See documentations of particular modules for details of their behaviors in training/evaluation mode, if they are affected, e.g. `Dropout`, `BatchNorm`, etc.

Parameters **mode** (*bool*) – whether to set training mode (`True`) or evaluation mode (`False`). Default: `True`.

Returns self

Return type Module

training: bool

type(*dst_type: Union[torch.dtype, str]*) → torch.nn.modules.module.T
Casts all parameters and buffers to `dst_type`.

Note: This method modifies the module in-place.

Parameters **dst_type** (*type or string*) – the desired type

Returns self

Return type Module

xpu(*device: Optional[Union[int, torch.device]] = None*) → torch.nn.modules.module.T
Moves all model parameters and buffers to the XPU.

This also makes associated parameters and buffers different objects. So it should be called before constructing optimizer if the module will live on XPU while being optimized.

Note: This method modifies the module in-place.

Parameters **device** (*int, optional*) – if specified, all parameters will be copied to that device

Returns self

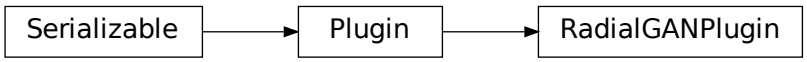
Return type Module

zero_grad(*set_to_none: bool = False*) → None
Sets gradients of all model parameters to zero. See similar function under `torch.optim.Optimizer` for more context.

Parameters **set_to_none** (*bool*) – instead of setting to zero, set the grads to None. See `torch.optim.Optimizer.zero_grad()` for details.

class RadialGANPlugin(*n_iter: int = 2000*, *generator_n_layers_hidden: int = 2*, *generator_n_units_hidden: int = 500*, *generator_nonlin: str = 'relu'*, *generator_dropout: float = 0.1*, *generator_opt_betas: tuple = (0.5, 0.999)*, *discriminator_n_layers_hidden: int = 2*, *discriminator_n_units_hidden: int = 500*, *discriminator_nonlin: str = 'leaky_relu'*, *discriminator_n_iter: int = 1*, *discriminator_dropout: float = 0.1*, *discriminator_opt_betas: tuple = (0.5, 0.999)*, *lr: float = 0.001*, *weight_decay: float = 0.001*, *batch_size: int = 500*, *random_state: int = 0*, *clipping_value: int = 1*, *lambda_gradient_penalty: float = 10*, *encoder_max_clusters: int = 10*, *device: Any = device(type='cpu')*, ***kwargs: Any*)

Bases: *synthcity.plugins.core.plugin.Plugin*



RadialGAN PyTorch implementation: Leveraging multiple datasets to improve target-specific predictive models using Generative Adversarial Networks.

Parameters

- **generator_n_layers_hidden** – int Number of hidden layers in the generator
- **generator_n_units_hidden** – int Number of hidden units in each layer of the Generator
- **generator_nonlin** – string, default 'leaky_relu' Nonlinearity to use in the generator. Can be 'elu', 'relu', 'selu' or 'leaky_relu'.
- **n_iter** – int Maximum number of iterations in the Generator.
- **generator_dropout** – float Dropout value. If 0, the dropout is not used.
- **discriminator_n_layers_hidden** – int Number of hidden layers in the discriminator
- **discriminator_n_units_hidden** – int Number of hidden units in each layer of the discriminator
- **discriminator_nonlin** – string, default 'leaky_relu' Nonlinearity to use in the discriminator. Can be 'elu', 'relu', 'selu' or 'leaky_relu'.
- **discriminator_n_iter** – int Maximum number of iterations in the discriminator.
- **discriminator_dropout** – float Dropout value for the discriminator. If 0, the dropout is not used.
- **lr** – float learning rate for optimizer.
- **weight_decay** – float l2 (ridge) penalty for the weights.
- **batch_size** – int Batch size
- **random_state** – int random seed to use
- **clipping_value** – int, default 0 Gradients clipping value. Zero disables the feature
- **encoder_max_clusters** – int The max number of clusters to create for continuous columns when encoding

Example

```
>>> import numpy as np
>>> from sklearn.datasets import load_iris
>>> from synthcity.plugins import Plugins
>>> from synthcity.plugins.core.dataloader import GenericDataLoader
>>>
>>> X, y = load_iris(as_frame = True, return_X_y = True)
>>> X["target"] = y
>>> X["domain"] = np.random.choice([0, 1], len(X)) # simulate domains
>>> dataloader = GenericDataLoader(X, domain_column="domain")
>>>
>>> plugin = Plugins().get("radialgan", n_iter = 100)
>>> plugin.fit(dataloader)
>>>
>>> plugin.generate(50)
```

class Config

Bases: object

arbitrary_types_allowed = True

validate_assignment = True

fit(*X: Union[*synthcity.plugins.core.dataloader.DataLoader*, pandas.core.frame.DataFrame]*, **args: Any*, ***kwargs: Any*) → Any

Training method the synthetic data plugin.

Parameters

- **X** – DataLoader. The reference dataset.
- **cond** – Optional, Union[pd.DataFrame, pd.Series, np.ndarray] Optional Training Conditional. The training conditional can be used to control to output of some models, like GANs or VAEs. The content can be anything, as long as it maps to the training dataset X. Usage example:

```
>>> from sklearn.datasets import load_iris
>>> from synthcity.plugins.core.dataloader import␣
↪GenericDataLoader
>>> from synthcity.plugins.core.constraints import Constraints
>>>
>>> # Load in `test_plugin` the generative model of choice
>>> # ....
>>>
>>> X, y = load_iris(as_frame=True, return_X_y=True)
>>> X["target"] = y
>>>
>>> X = GenericDataLoader(X)
>>> test_plugin.fit(X, cond=y)
>>>
>>> count = 10
>>> X_gen = test_plugin.generate(count, cond=np.ones(count))
>>>
>>> # The Conditional only optimizes the output generation
>>> # for GANs and VAEs, but does NOT guarantee the samples
```

(continues on next page)

(continued from previous page)

```
>>> # are only from that condition.
>>> # If you want to guarantee that output contains only
>>> # "target" == 1 samples, use Constraints.
>>>
>>> constraints = Constraints(
>>>     rules=[
>>>         ("target", "==", 1),
>>>     ]
>>> )
>>> X_gen = test_plugin.generate(count,
>>>         cond=np.ones(count),
>>>         constraints=constraints
>>>        )
>>> assert (X_gen["target"] == 1).all()
```

Returns self

classmethod fqdn() → str
The Fully-Qualified name of the plugin.

generate(*count: Optional[int] = None*, *constraints: Optional[*synthcity.plugins.core.constraints.Constraints*] = None*, *random_state: Optional[int] = None*, ***kwargs: Any*) → *synthcity.plugins.core.dataloader.DataLoader*
Synthetic data generation method.

Parameters

- **count** – optional int. The number of samples to generate. If None, it generated len(reference_dataset) samples.
- **cond** – Optional, Union[pd.DataFrame, pd.Series, np.ndarray]. Optional Generation Conditional. The conditional can be used only if the model was trained using a conditional too. If provided, it must have *count* length. Not all models support conditionals. The conditionals can be used in VAEs or GANs to speed-up the generation under some constraints. For model agnostic solutions, check out the *constraints* parameter.
- **constraints** – optional Constraints. Optional constraints to apply on the generated data. If none, the reference schema constraints are applied. The constraints are model agnostic, and will filter the output of the generative model. The constraints are a list of rules. Each rule is a tuple of the form (<feature>, <operation>, <value>).

  Valid Operations:

  - ”<”, “lt” : less than <value>
  - ”<=”, “le”: less or equal with <value>
  - ”>”, “gt” : greater than <value>
  - ”>=”, “ge”: greater or equal with <value>
  - ”==”, “eq”: equal with <value>
  - ”in”: valid for categorical features, and <value> must be array. for example, (“target”, “in”, [0, 1])
  - ”dtype”: <value> can be a data type. For example, (“target”, “dtype”, “int”)

  Usage example:

```
>>> from synthcity.plugins.core.constraints import Constraints
>>> constraints = Constraints(
>>>   rules=[
>>>             ("InterestingFeature", "==", 0),
>>>         ]
>>>     )
>>>
>>> syn_data = syn_model.generate(
        count=count,
        constraints=constraints
    ).dataframe()
>>>
>>> assert (syn_data["InterestingFeature"] == 0).all()
```

- **random_state** – optional int. Optional random seed to use.

Returns <count> synthetic samples

static hyperparameter_space(***kwargs: Any*) → List[*synthcity.plugins.core.distribution.Distribution*]
Returns the hyperparameter space for the derived plugin.

static load(*buff: bytes*) → Any

static load_dict(*representation: dict*) → Any

static name() → str
The name of the plugin.

plot(*plt: Any*, *X:* synthcity.plugins.core.dataloader.DataLoader, *count: Optional[int] = None*, *plots: list = ['marginal', 'associations', 'tsne']*, ***kwargs: Any*) → Any
Plot the real-synthetic distributions.

Parameters

- **plt** – output
- **X** – DataLoader. The reference dataset.

Returns self

classmethod sample_hyperparameters(**args: Any*, ***kwargs: Any*) → Dict[str, Any]
Sample value from the hyperparameter space for the current plugin.

classmethod sample_hyperparameters_optuna(*trial: Any*, **args: Any*, ***kwargs: Any*) → Dict[str, Any]

save() → bytes

save_dict() → dict

save_to_file(*path: pathlib.Path*) → bytes

schema() → *synthcity.plugins.core.schema.Schema*
The reference schema

schema_includes(*other: Union[*synthcity.plugins.core.dataloader.DataLoader, *pandas.core.frame.DataFrame]*) → bool
Helper method to test if the reference schema includes a Dataset

Parameters **other** – DataLoader. The dataset to test

Returns bool, if the schema includes the dataset or not.

training_schema() → *synthcity.plugins.core.schema.Schema*
The internal schema

static type() → str
The type of the plugin.

static version() → str
API version

class TabularRadialGAN(*X: pandas.core.frame.DataFrame*, *domain_column: str*, *n_units_latent: int*, *generator_n_layers_hidden: int = 2*, *generator_n_units_hidden: int = 150*, *generator_nonlin: str = 'leaky_relu'*, *generator_nonlin_out_discrete: str = 'softmax'*, *generator_nonlin_out_continuous: str = 'none'*, *generator_n_iter: int = 1000*, *generator_dropout: float = 0.01*, *generator_lr: float = 0.001*, *generator_weight_decay: float = 0.001*, *generator_opt_betas: tuple = (0.9, 0.999)*, *discriminator_n_layers_hidden: int = 3*, *discriminator_n_units_hidden: int = 300*, *discriminator_nonlin: str = 'leaky_relu'*, *discriminator_n_iter: int = 1*, *discriminator_dropout: float = 0.1*, *discriminator_lr: float = 0.001*, *discriminator_weight_decay: float = 0.001*, *discriminator_opt_betas: tuple = (0.9, 0.999)*, *batch_size: int = 64*, *n_iter_print: int = 100*, *random_state: int = 0*, *n_iter_min: int = 100*, *clipping_value: int = 0*, *lambda_gradient_penalty: float = 10*, *encoder_max_clusters: int = 20*, *device: Any = device(type='cpu')*)
Bases: `torch.nn.modules.module.Module`

RadialGAN for tabular data.

This class combines RadialGAN and tabular encoder to form a generative model for tabular data.

Parameters

- **X** – pd.DataFrame Input dataset
- **domain_column** – str The domain column
- **n_units_in** – int Number of features
- **generator_n_layers_hidden** – int Number of hidden layers in the generator
- **generator_n_units_hidden** – int Number of hidden units in each layer of the Generator
- **generator_nonlin** – string, default ‘elu’ Nonlinearity to use in the generator. Can be ‘elu’, ‘relu’, ‘selu’ or ‘leaky_relu’.
- **generator_n_iter** – int Maximum number of iterations in the Generator.
- **generator_dropout** – float Dropout value. If 0, the dropout is not used.
- **discriminator_n_layers_hidden** – int Number of hidden layers in the discriminator
- **discriminator_n_units_hidden** – int Number of hidden units in each layer of the discriminator
- **discriminator_nonlin** – string, default ‘relu’ Nonlinearity to use in the discriminator. Can be ‘elu’, ‘relu’, ‘selu’ or ‘leaky_relu’.
- **discriminator_n_iter** – int Maximum number of iterations in the discriminator.
- **discriminator_dropout** – float Dropout value for the discriminator. If 0, the dropout is not used.
- **lr** – float learning rate for optimizer..
- **weight_decay** – float l2 (ridge) penalty for the weights.

- **batch_size** – int Batch size
- **n_iter_print** – int Number of iterations after which to print updates and check the validation loss.
- **random_state** – int random_state used
- **clipping_value** – int, default 0 Gradients clipping value
- **lambda_gradient_penalty** – float Lambda weight for the gradient penalty
- **encoder_max_clusters** – int The max number of clusters to create for continuous columns when encoding

T_destination
alias of TypeVar('T_destination', bound=`Dict[str, Any]`)

add_module(*name: str*, *module: Optional[torch.nn.modules.module.Module]*) → None
Adds a child module to the current module.

The module can be accessed as an attribute using the given name.

Parameters

- **name** (*str*) – name of the child module. The child module can be accessed from this module using the given name
- **module** (*Module*) – child module to be added to the module.

apply(*fn: Callable[[torch.nn.modules.module.Module], None]*) → torch.nn.modules.module.T
Applies `fn` recursively to every submodule (as returned by `.children()`) as well as self. Typical use includes initializing the parameters of a model (see also nn-init-doc).

Parameters **fn** (`Module -> None`) – function to be applied to each submodule

Returns self

Return type Module

Example:

```
>>> @torch.no_grad()
>>> def init_weights(m):
>>>     print(m)
>>>     if type(m) == nn.Linear:
>>>         m.weight.fill_(1.0)
>>>         print(m.weight)
>>> net = nn.Sequential(nn.Linear(2, 2), nn.Linear(2, 2))
>>> net.apply(init_weights)
Linear(in_features=2, out_features=2, bias=True)
Parameter containing:
tensor([[1., 1.],
        [1., 1.]], requires_grad=True)
Linear(in_features=2, out_features=2, bias=True)
Parameter containing:
tensor([[1., 1.],
        [1., 1.]], requires_grad=True)
Sequential(
  (0): Linear(in_features=2, out_features=2, bias=True)
  (1): Linear(in_features=2, out_features=2, bias=True)
)
```

bfloat16() → torch.nn.modules.module.T
Casts all floating point parameters and buffers to `bfloat16` datatype.

Note: This method modifies the module in-place.

Returns self

Return type Module

buffers(*recurse: bool = True*) → Iterator[torch.Tensor]
Returns an iterator over module buffers.

Parameters **recurse** (*bool*) – if True, then yields buffers of this module and all submodules. Otherwise, yields only buffers that are direct members of this module.

Yields *torch.Tensor* – module buffer

Example:

```
>>> # xdoctest: +SKIP("undefined vars")
>>> for buf in model.buffers():
>>>     print(type(buf), buf.size())
<class 'torch.Tensor'> (20L,)
<class 'torch.Tensor'> (20L, 1L, 5L, 5L)
```

children() → Iterator[torch.nn.modules.module.Module]
Returns an iterator over immediate children modules.

Yields *Module* – a child module

cpu() → torch.nn.modules.module.T
Moves all model parameters and buffers to the CPU.

Note: This method modifies the module in-place.

Returns self

Return type Module

cuda(*device: Optional[Union[int, torch.device]] = None*) → torch.nn.modules.module.T
Moves all model parameters and buffers to the GPU.

This also makes associated parameters and buffers different objects. So it should be called before constructing optimizer if the module will live on GPU while being optimized.

Note: This method modifies the module in-place.

Parameters **device** (*int, optional*) – if specified, all parameters will be copied to that device

Returns self

Return type Module

decode(*X: pandas.core.frame.DataFrame*) → pandas.core.frame.DataFrame

double() → torch.nn.modules.module.T
Casts all floating point parameters and buffers to double datatype.

Note: This method modifies the module in-place.

Returns self

Return type Module

dump_patches: bool = False

encode(*X: pandas.core.frame.DataFrame*) → pandas.core.frame.DataFrame

eval() → torch.nn.modules.module.T
Sets the module in evaluation mode.

This has any effect only on certain modules. See documentations of particular modules for details of their behaviors in training/evaluation mode, if they are affected, e.g. Dropout, BatchNorm, etc.

This is equivalent with self.train(False).

See locally-disable-grad-doc for a comparison between *.eval()* and several similar mechanisms that may be confused with it.

Returns self

Return type Module

extra_repr() → str
Set the extra representation of the module

To print customized extra information, you should re-implement this method in your own modules. Both single-line and multi-line strings are acceptable.

fit(*X: pandas.core.frame.DataFrame*) → Any

float() → torch.nn.modules.module.T
Casts all floating point parameters and buffers to float datatype.

Note: This method modifies the module in-place.

Returns self

Return type Module

forward(*count: int*, *domains: Optional[List[int]] = None*) → torch.Tensor
Defines the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the Module instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

generate(*count: int*, *domains: Optional[List[int]] = None*) → pandas.core.frame.DataFrame

get_buffer(*target: str*) → torch.Tensor
Returns the buffer given by `target` if it exists, otherwise throws an error.

See the docstring for `get_submodule` for a more detailed explanation of this method's functionality as well as how to correctly specify `target`.

Parameters **target** – The fully-qualified string name of the buffer to look for. (See `get_submodule` for how to specify a fully-qualified string.)

Returns The buffer referenced by `target`

Return type torch.Tensor

Raises **AttributeError** – If the target string references an invalid path or resolves to something that is not a buffer

get_extra_state() → Any
Returns any extra state to include in the module's state_dict. Implement this and a corresponding *set_extra_state()* for your module if you need to store extra state. This function is called when building the module's *state_dict()*.

Note that extra state should be pickleable to ensure working serialization of the state_dict. We only provide provide backwards compatibility guarantees for serializing Tensors; other objects may break backwards compatibility if their serialized pickled form changes.

Returns Any extra state to store in the module's state_dict

Return type object

get_parameter(*target: str*) → torch.nn.parameter.Parameter
Returns the parameter given by `target` if it exists, otherwise throws an error.

See the docstring for `get_submodule` for a more detailed explanation of this method's functionality as well as how to correctly specify `target`.

Parameters **target** – The fully-qualified string name of the Parameter to look for. (See `get_submodule` for how to specify a fully-qualified string.)

Returns The Parameter referenced by `target`

Return type torch.nn.Parameter

Raises **AttributeError** – If the target string references an invalid path or resolves to something that is not an `nn.Parameter`

get_submodule(*target: str*) → torch.nn.modules.module.Module
Returns the submodule given by `target` if it exists, otherwise throws an error.

For example, let's say you have an `nn.Module` `A` that looks like this:

```
A(
    (net_b): Module(
        (net_c): Module(
            (conv): Conv2d(16, 33, kernel_size=(3, 3), stride=(2, 2))
        )
        (linear): Linear(in_features=100, out_features=200, bias=True)
    )
)
```

(The diagram shows an `nn.Module` `A`. `A` has a nested submodule `net_b`, which itself has two submodules `net_c` and `linear`. `net_c` then has a submodule `conv`.)

To check whether or not we have the `linear` submodule, we would call `get_submodule("net_b.linear")`. To check whether we have the `conv` submodule, we would call `get_submodule("net_b.net_c.conv")`.

The runtime of `get_submodule` is bounded by the degree of module nesting in `target`. A query against `named_modules` achieves the same result, but it is O(N) in the number of transitive modules. So, for a simple check to see if some submodule exists, `get_submodule` should always be used.

Parameters `target` – The fully-qualified string name of the submodule to look for. (See above example for how to specify a fully-qualified string.)

Returns The submodule referenced by `target`

Return type torch.nn.Module

Raises `AttributeError` – If the target string references an invalid path or resolves to something that is not an `nn.Module`

`half`() → torch.nn.modules.module.T

Casts all floating point parameters and buffers to `half` datatype.

Note: This method modifies the module in-place.

Returns self

Return type Module

`ipu`(*device: Optional[Union[int, torch.device]] = None*) → torch.nn.modules.module.T

Moves all model parameters and buffers to the IPU.

This also makes associated parameters and buffers different objects. So it should be called before constructing optimizer if the module will live on IPU while being optimized.

Note: This method modifies the module in-place.

Parameters `device` (*int, optional*) – if specified, all parameters will be copied to that device

Returns self

Return type Module

`load_state_dict`(*state_dict: Mapping[str, Any]*, *strict: bool = True*)

Copies parameters and buffers from *state_dict* into this module and its descendants. If `strict` is `True`, then the keys of *state_dict* must exactly match the keys returned by this module's `state_dict()` function.

Parameters

- `state_dict` (*dict*) – a dict containing parameters and persistent buffers.
- `strict` (*bool, optional*) – whether to strictly enforce that the keys in *state_dict* match the keys returned by this module's `state_dict()` function. Default: `True`

Returns

- **missing_keys** is a list of str containing the missing keys
- **unexpected_keys** is a list of str containing the unexpected keys

Return type NamedTuple with missing_keys and unexpected_keys fields

---

Note: If a parameter or buffer is registered as None and its corresponding key exists in *state_dict*, *load_state_dict()* will raise a RuntimeError.

---

modules() → Iterator[torch.nn.modules.module.Module]
Returns an iterator over all modules in the network.

Yields *Module* – a module in the network

Note: Duplicate modules are returned only once. In the following example, l will be returned only once.

Example:

```
>>> l = nn.Linear(2, 2)
>>> net = nn.Sequential(l, l)
>>> for idx, m in enumerate(net.modules()):
...     print(idx, '->', m)

0 -> Sequential(
  (0): Linear(in_features=2, out_features=2, bias=True)
  (1): Linear(in_features=2, out_features=2, bias=True)
)
1 -> Linear(in_features=2, out_features=2, bias=True)
```

named_buffers(*prefix: str = ''*, *recurse: bool = True*) → Iterator[Tuple[str, torch.Tensor]]
Returns an iterator over module buffers, yielding both the name of the buffer as well as the buffer itself.

Parameters

- **prefix** (*str*) – prefix to prepend to all buffer names.
- **recurse** (*bool*) – if True, then yields buffers of this module and all submodules. Otherwise, yields only buffers that are direct members of this module.

Yields *(str, torch.Tensor)* – Tuple containing the name and buffer

Example:

```
>>> # xdoctest: +SKIP("undefined vars")
>>> for name, buf in self.named_buffers():
>>>    if name in ['running_var']:
>>>        print(buf.size())
```

named_children() → Iterator[Tuple[str, torch.nn.modules.module.Module]]
Returns an iterator over immediate children modules, yielding both the name of the module as well as the module itself.

Yields *(str, Module)* – Tuple containing a name and child module

Example:

```
>>> # xdoctest: +SKIP("undefined vars")
>>> for name, module in model.named_children():
>>>     if name in ['conv4', 'conv5']:
>>>         print(module)
```

named_modules(*memo: Optional[Set[torch.nn.modules.module.Module]] = None*, *prefix: str = ''*, *remove_duplicate: bool = True*)

Returns an iterator over all modules in the network, yielding both the name of the module as well as the module itself.

Parameters

- **memo** – a memo to store the set of modules already added to the result
- **prefix** – a prefix that will be added to the name of the module
- **remove_duplicate** – whether to remove the duplicated module instances in the result or not

Yields *(str, Module)* – Tuple of name and module

Note: Duplicate modules are returned only once. In the following example, `l` will be returned only once.

Example:

```
>>> l = nn.Linear(2, 2)
>>> net = nn.Sequential(l, l)
>>> for idx, m in enumerate(net.named_modules()):
...     print(idx, '->', m)

0 -> ('', Sequential(
  (0): Linear(in_features=2, out_features=2, bias=True)
  (1): Linear(in_features=2, out_features=2, bias=True)
))
1 -> ('0', Linear(in_features=2, out_features=2, bias=True))
```

named_parameters(*prefix: str = ''*, *recurse: bool = True*) → Iterator[Tuple[str, torch.nn.parameter.Parameter]]

Returns an iterator over module parameters, yielding both the name of the parameter as well as the parameter itself.

Parameters

- **prefix** (*str*) – prefix to prepend to all parameter names.
- **recurse** (*bool*) – if True, then yields parameters of this module and all submodules. Otherwise, yields only parameters that are direct members of this module.

Yields *(str, Parameter)* – Tuple containing the name and parameter

Example:

```
>>> # xdoctest: +SKIP("undefined vars")
>>> for name, param in self.named_parameters():
>>>    if name in ['bias']:
>>>        print(param.size())
```

parameters(*recurse: bool = True*) → Iterator[torch.nn.parameter.Parameter]

Returns an iterator over module parameters.

This is typically passed to an optimizer.

Parameters **recurse** (*bool*) – if True, then yields parameters of this module and all submodules. Otherwise, yields only parameters that are direct members of this module.

Yields *Parameter* – module parameter

Example:

```
>>> # xdoctest: +SKIP("undefined vars")
>>> for param in model.parameters():
>>>     print(type(param), param.size())
<class 'torch.Tensor'> (20L,)
<class 'torch.Tensor'> (20L, 1L, 5L, 5L)
```

register_backward_hook(*hook: Callable[[torch.nn.modules.module.Module, Union[Tuple[torch.Tensor, ...], torch.Tensor], Union[Tuple[torch.Tensor, ...], torch.Tensor]], Union[None, torch.Tensor]]*) → torch.utils.hooks.RemovableHandle

Registers a backward hook on the module.

This function is deprecated in favor of `register_full_backward_hook()` and the behavior of this function will change in future versions.

Returns a handle that can be used to remove the added hook by calling `handle.remove()`

Return type `torch.utils.hooks.RemovableHandle`

register_buffer(*name: str*, *tensor: Optional[torch.Tensor]*, *persistent: bool = True*) → None

Adds a buffer to the module.

This is typically used to register a buffer that should not to be considered a model parameter. For example, BatchNorm's `running_mean` is not a parameter, but is part of the module's state. Buffers, by default, are persistent and will be saved alongside parameters. This behavior can be changed by setting `persistent` to `False`. The only difference between a persistent buffer and a non-persistent buffer is that the latter will not be a part of this module's *state_dict*.

Buffers can be accessed as attributes using given names.

Parameters

- **name** (*str*) – name of the buffer. The buffer can be accessed from this module using the given name
- **tensor** (*Tensor or None*) – buffer to be registered. If `None`, then operations that run on buffers, such as *cuda*, are ignored. If `None`, the buffer is **not** included in the module's *state_dict*.
- **persistent** (*bool*) – whether the buffer is part of this module's *state_dict*.

Example:

```
>>> # xdoctest: +SKIP("undefined vars")
>>> self.register_buffer('running_mean', torch.zeros(num_features))
```

register_forward_hook(*hook: Callable[[...], None]*) → torch.utils.hooks.RemovableHandle

Registers a forward hook on the module.

The hook will be called every time after *forward()* has computed an output. It should have the following signature:

```
hook(module, input, output) -> None or modified output
```

The input contains only the positional arguments given to the module. Keyword arguments won't be passed to the hooks and only to the `forward`. The hook can modify the output. It can modify the input inplace but it will not have effect on forward since this is called after *forward()* is called.

Returns a handle that can be used to remove the added hook by calling `handle.remove()`

Return type torch.utils.hooks.RemovableHandle

register_forward_pre_hook(*hook: Callable[[...], None]*) → torch.utils.hooks.RemovableHandle

Registers a forward pre-hook on the module.

The hook will be called every time before *forward()* is invoked. It should have the following signature:

```
hook(module, input) -> None or modified input
```

The input contains only the positional arguments given to the module. Keyword arguments won't be passed to the hooks and only to the `forward`. The hook can modify the input. User can either return a tuple or a single modified value in the hook. We will wrap the value into a tuple if a single value is returned(unless that value is already a tuple).

Returns a handle that can be used to remove the added hook by calling `handle.remove()`

Return type torch.utils.hooks.RemovableHandle

register_full_backward_hook(*hook: Callable[[torch.nn.modules.module.Module, Union[Tuple[torch.Tensor, ...], torch.Tensor], Union[Tuple[torch.Tensor, ...], torch.Tensor]], Union[None, torch.Tensor]]*) → torch.utils.hooks.RemovableHandle

Registers a backward hook on the module.

The hook will be called every time the gradients with respect to a module are computed, i.e. the hook will execute if and only if the gradients with respect to module outputs are computed. The hook should have the following signature:

```
hook(module, grad_input, grad_output) -> tuple(Tensor) or None
```

The `grad_input` and `grad_output` are tuples that contain the gradients with respect to the inputs and outputs respectively. The hook should not modify its arguments, but it can optionally return a new gradient with respect to the input that will be used in place of `grad_input` in subsequent computations. `grad_input` will only correspond to the inputs given as positional arguments and all kwarg arguments are ignored. Entries in `grad_input` and `grad_output` will be `None` for all non-Tensor arguments.

For technical reasons, when this hook is applied to a Module, its forward function will receive a view of each Tensor passed to the Module. Similarly the caller will receive a view of each Tensor returned by the Module's forward function.

Warning: Modifying inputs or outputs inplace is not allowed when using backward hooks and will raise an error.

Returns a handle that can be used to remove the added hook by calling `handle.remove()`

Return type torch.utils.hooks.RemovableHandle

register_load_state_dict_post_hook(*hook*)

Registers a post hook to be run after module's `load_state_dict` is called.

It should have the following signature:: hook(module, incompatible_keys) -> None

The `module` argument is the current module that this hook is registered on, and the `incompatible_keys` argument is a `NamedTuple` consisting of attributes `missing_keys` and `unexpected_keys`. `missing_keys` is a `list` of `str` containing the missing keys and `unexpected_keys` is a `list` of `str` containing the unexpected keys.

The given incompatible_keys can be modified inplace if needed.

Note that the checks performed when calling *load_state_dict()* with `strict=True` are affected by modifications the hook makes to `missing_keys` or `unexpected_keys`, as expected. Additions to either set of keys will result in an error being thrown when `strict=True`, and clearning out both missing and unexpected keys will avoid an error.

Returns a handle that can be used to remove the added hook by calling `handle.remove()`

Return type `torch.utils.hooks.RemovableHandle`

register_module(*name: str*, *module: Optional[torch.nn.modules.module.Module]*) → None
Alias for *add_module()*.

register_parameter(*name: str*, *param: Optional[torch.nn.parameter.Parameter]*) → None
Adds a parameter to the module.

The parameter can be accessed as an attribute using given name.

Parameters

- **name** (*str*) – name of the parameter. The parameter can be accessed from this module using the given name
- **param** (*Parameter or None*) – parameter to be added to the module. If `None`, then operations that run on parameters, such as *cuda*, are ignored. If `None`, the parameter is **not** included in the module's *state_dict*.

requires_grad_(*requires_grad: bool = True*) → torch.nn.modules.module.T
Change if autograd should record operations on parameters in this module.

This method sets the parameters' `requires_grad` attributes in-place.

This method is helpful for freezing part of the module for finetuning or training parts of a model individually (e.g., GAN training).

See locally-disable-grad-doc for a comparison between *.requires_grad_()* and several similar mechanisms that may be confused with it.

Parameters **requires_grad** (*bool*) – whether autograd should record operations on parameters in this module. Default: `True`.

Returns self

Return type Module

set_extra_state(*state: Any*)
This function is called from *load_state_dict()* to handle any extra state found within the *state_dict*. Implement this function and a corresponding *get_extra_state()* for your module if you need to store extra state within its *state_dict*.

Parameters **state** (*dict*) – Extra state from the *state_dict*

share_memory() → torch.nn.modules.module.T
See `torch.Tensor.share_memory_()`

state_dict(**args*, *destination=None*, *prefix=''*, *keep_vars=False*)
Returns a dictionary containing references to the whole state of the module.

Both parameters and persistent buffers (e.g. running averages) are included. Keys are corresponding parameter and buffer names. Parameters and buffers set to `None` are not included.

Note: The returned object is a shallow copy. It contains references to the module's parameters and buffers.

> **Warning:** Currently `state_dict()` also accepts positional arguments for `destination`, `prefix` and `keep_vars` in order. However, this is being deprecated and keyword arguments will be enforced in future releases.

> **Warning:** Please avoid the use of argument `destination` as it is not designed for end-users.

Parameters

- **destination** (*dict, optional*) – If provided, the state of module will be updated into the dict and the same object is returned. Otherwise, an `OrderedDict` will be created and returned. Default: `None`.
- **prefix** (*str, optional*) – a prefix added to parameter and buffer names to compose the keys in state_dict. Default: `''`.
- **keep_vars** (*bool, optional*) – by default the `Tensor` s returned in the state dict are detached from autograd. If it's set to `True`, detaching will not be performed. Default: `False`.

Returns a dictionary containing a whole state of the module

Return type dict

Example:

```
>>> # xdoctest: +SKIP("undefined vars")
>>> module.state_dict().keys()
['bias', 'weight']
```

to(**args*, ***kwargs*)

Moves and/or casts the parameters and buffers.

This can be called as

to(*device=None*, *dtype=None*, *non_blocking=False*)

to(*dtype*, *non_blocking=False*)

to(*tensor*, *non_blocking=False*)

to(*memory_format=torch.channels_last*)

Its signature is similar to `torch.Tensor.to()`, but only accepts floating point or complex `dtype`s. In addition, this method will only cast the floating point or complex parameters and buffers to `dtype` (if given). The integral parameters and buffers will be moved `device`, if that is given, but with dtypes unchanged. When `non_blocking` is set, it tries to convert/move asynchronously with respect to the host if possible, e.g., moving CPU Tensors with pinned memory to CUDA devices.

See below for examples.

> **Note:** This method modifies the module in-place.

Parameters

- **device** (`torch.device`) – the desired device of the parameters and buffers in this module

- **dtype** (torch.dtype) – the desired floating point or complex dtype of the parameters and buffers in this module
- **tensor** (*torch.Tensor*) – Tensor whose dtype and device are the desired dtype and device for all parameters and buffers in this module
- **memory_format** (torch.memory_format) – the desired memory format for 4D parameters and buffers in this module (keyword only argument)

Returns self

Return type Module

Examples:

```
>>> # xdoctest: +IGNORE_WANT("non-deterministic")
>>> linear = nn.Linear(2, 2)
>>> linear.weight
Parameter containing:
tensor([[ 0.1913, -0.3420],
        [-0.5113, -0.2325]])
>>> linear.to(torch.double)
Linear(in_features=2, out_features=2, bias=True)
>>> linear.weight
Parameter containing:
tensor([[ 0.1913, -0.3420],
        [-0.5113, -0.2325]], dtype=torch.float64)
>>> # xdoctest: +REQUIRES(env:TORCH_DOCTEST_CUDA1)
>>> gpu1 = torch.device("cuda:1")
>>> linear.to(gpu1, dtype=torch.half, non_blocking=True)
Linear(in_features=2, out_features=2, bias=True)
>>> linear.weight
Parameter containing:
tensor([[ 0.1914, -0.3420],
        [-0.5112, -0.2324]], dtype=torch.float16, device='cuda:1')
>>> cpu = torch.device("cpu")
>>> linear.to(cpu)
Linear(in_features=2, out_features=2, bias=True)
>>> linear.weight
Parameter containing:
tensor([[ 0.1914, -0.3420],
        [-0.5112, -0.2324]], dtype=torch.float16)

>>> linear = nn.Linear(2, 2, bias=None).to(torch.cdouble)
>>> linear.weight
Parameter containing:
tensor([[ 0.3741+0.j,  0.2382+0.j],
        [ 0.5593+0.j, -0.4443+0.j]], dtype=torch.complex128)
>>> linear(torch.ones(3, 2, dtype=torch.cdouble))
tensor([[0.6122+0.j, 0.1150+0.j],
        [0.6122+0.j, 0.1150+0.j],
        [0.6122+0.j, 0.1150+0.j]], dtype=torch.complex128)
```

to_empty(*, *device: Union[str, torch.device]*) → torch.nn.modules.module.T

Moves the parameters and buffers to the specified device without copying storage.

Parameters **device** (torch.device) – The desired device of the parameters and buffers in this module.

Returns self

Return type Module

train(*mode: bool = True*) → torch.nn.modules.module.T
Sets the module in training mode.

This has any effect only on certain modules. See documentations of particular modules for details of their behaviors in training/evaluation mode, if they are affected, e.g. Dropout, BatchNorm, etc.

Parameters **mode** (*bool*) – whether to set training mode (True) or evaluation mode (False). Default: True.

Returns self

Return type Module

training: bool

type(*dst_type: Union[torch.dtype, str]*) → torch.nn.modules.module.T
Casts all parameters and buffers to dst_type.

Note: This method modifies the module in-place.

Parameters **dst_type** (*type or string*) – the desired type

Returns self

Return type Module

xpu(*device: Optional[Union[int, torch.device]] = None*) → torch.nn.modules.module.T
Moves all model parameters and buffers to the XPU.

This also makes associated parameters and buffers different objects. So it should be called before constructing optimizer if the module will live on XPU while being optimized.

Note: This method modifies the module in-place.

Parameters **device** (*int, optional*) – if specified, all parameters will be copied to that device

Returns self

Return type Module

zero_grad(*set_to_none: bool = False*) → None
Sets gradients of all model parameters to zero. See similar function under torch.optim.Optimizer for more context.

Parameters **set_to_none** (*bool*) – instead of setting to zero, set the grads to None. See torch.optim.Optimizer.zero_grad() for details.

plugin
alias of *synthcity.plugins.domain_adaptation.plugin_radialgan.RadialGANPlugin*

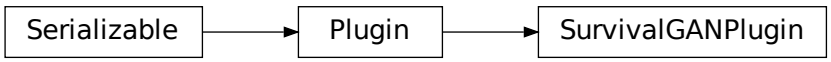
10.1.4 Static Survival Analysis

synthcity.plugins.survival_analysis.plugin_survival_gan module

Reference: "SurvivalGAN: Generating time-to-event Data for Survival Analysis", B. Cebere*, A. Norcliffe*, F. Imrie, M. van der Schaar, AISTATS 2023

class SurvivalGANPlugin(*uncensoring_model: str = 'survival_function_regression'*, *dataloader_sampling_strategy: str = 'imbalanced_time_censoring'*, *tte_strategy: str = 'survival_function'*, *censoring_strategy: str = 'random'*, *device: Any = device(type='cpu')*, *use_survival_conditional: bool = True*, *workspace: pathlib.Path = PosixPath('workspace')*, *random_state: int = 0*, *compress_dataset: bool = False*, *sampling_patience: int = 500*, ***kwargs: Any*)

Bases: *synthcity.plugins.core.plugin.Plugin*



Survival Analysis Pipeline based on AdsGAN.

Parameters

- **uncensoring_model** – str The time-to-event model: "survival_function_regression".
- **dataloader_sampling_strategy** – str, default = imbalanced_time_censoring Training sampling strategy: none, imbalanced_censoring, imbalanced_time_censoring
- **tte_strategy** – str The time-to-event generation strategy: survival_function, uncensoring.
- **censoring_strategy** – str For the generated data, how to censor subjects: "random" or "covariate_dependent"
- **device** – torch device to use for training(cpu/cuda)
- **kwargs** – Any "adsgan" additional args, like n_iter = 100 etc.
- **arguments** (*# Core Plugin*) –
- **workspace** – Path. Optional Path for caching intermediary results.
- **compress_dataset** – bool. Default = False. Drop redundant features before training the generator.
- **sampling_patience** – int. Max inference iterations to wait for the generated data to match the training schema.

Example

```
>>> from lifelines.datasets import load_rossi
>>> from synthcity.plugins import Plugins
>>> from synthcity.plugins.core.dataloader import SurvivalAnalysisDataLoader
>>>
>>> X = load_rossi()
>>> data = SurvivalAnalysisDataLoader(
>>>        X,
>>>        target_column="arrest",
>>>        time_to_event_column="week",
>>> )
>>>
>>> plugin = Plugins().get("survival_gan")
>>> plugin.fit(data)
>>>
>>> plugin.generate(count = 50)
```

class Config

Bases: object

arbitrary_types_allowed = True

validate_assignment = True

fit(*X: Union[*synthcity.plugins.core.dataloader.DataLoader*, pandas.core.frame.DataFrame]*, **args: Any*, ***kwargs: Any*) → Any

Training method the synthetic data plugin.

Parameters

- **X** – DataLoader. The reference dataset.
- **cond** – Optional, Union[pd.DataFrame, pd.Series, np.ndarray] Optional Training Conditional. The training conditional can be used to control to output of some models, like GANs or VAEs. The content can be anything, as long as it maps to the training dataset X. Usage example:

```
>>> from sklearn.datasets import load_iris
>>> from synthcity.plugins.core.dataloader import␣
↪GenericDataLoader
>>> from synthcity.plugins.core.constraints import Constraints
>>>
>>> # Load in `test_plugin` the generative model of choice
>>> # ....
>>>
>>> X, y = load_iris(as_frame=True, return_X_y=True)
>>> X["target"] = y
>>>
>>> X = GenericDataLoader(X)
>>> test_plugin.fit(X, cond=y)
>>>
>>> count = 10
>>> X_gen = test_plugin.generate(count, cond=np.ones(count))
>>>
>>> # The Conditional only optimizes the output generation
```

(continues on next page)

(continued from previous page)

```
>>> # for GANs and VAEs, but does NOT guarantee the samples
>>> # are only from that condition.
>>> # If you want to guarantee that output contains only
>>> # "target" == 1 samples, use Constraints.
>>>
>>> constraints = Constraints(
>>>     rules=[
>>>         ("target", "==", 1),
>>>     ]
>>> )
>>> X_gen = test_plugin.generate(count,
>>>         cond=np.ones(count),
>>>         constraints=constraints
>>>        )
>>> assert (X_gen["target"] == 1).all()
```

Returns self

classmethod fqdn() → str
The Fully-Qualified name of the plugin.

generate(*count: Optional[int] = None*, *constraints: Optional[*synthcity.plugins.core.constraints.Constraints*] = None*, *random_state: Optional[int] = None*, ***kwargs: Any*) → *synthcity.plugins.core.dataloader.DataLoader*
Synthetic data generation method.

Parameters

- **count** – optional int. The number of samples to generate. If None, it generated len(reference_dataset) samples.
- **cond** – Optional, Union[pd.DataFrame, pd.Series, np.ndarray]. Optional Generation Conditional. The conditional can be used only if the model was trained using a conditional too. If provided, it must have *count* length. Not all models support conditionals. The conditionals can be used in VAEs or GANs to speed-up the generation under some constraints. For model agnostic solutions, check out the *constraints* parameter.
- **constraints** – optional Constraints. Optional constraints to apply on the generated data. If none, the reference schema constraints are applied. The constraints are model agnostic, and will filter the output of the generative model. The constraints are a list of rules. Each rule is a tuple of the form (<feature>, <operation>, <value>).

  Valid Operations:

  - ”<”, “lt” : less than <value>
  - ”<=”, “le”: less or equal with <value>
  - ”>”, “gt” : greater than <value>
  - ”>=”, “ge”: greater or equal with <value>
  - ”==”, “eq”: equal with <value>
  - ”in”: valid for categorical features, and <value> must be array. for example, (“target”, “in”, [0, 1])
  - ”dtype”: <value> can be a data type. For example, (“target”, “dtype”, “int”)

  Usage example:

```
>>> from synthcity.plugins.core.constraints import Constraints
>>> constraints = Constraints(
>>>   rules=[
>>>             ("InterestingFeature", "==", 0),
>>>         ]
>>>     )
>>>
>>> syn_data = syn_model.generate(
        count=count,
        constraints=constraints
    ).dataframe()
>>>
>>> assert (syn_data["InterestingFeature"] == 0).all()
```

- **random_state** – optional int. Optional random seed to use.

Returns <count> synthetic samples

static hyperparameter_space(***kwargs: Any*) → List[*synthcity.plugins.core.distribution.Distribution*]
Returns the hyperparameter space for the derived plugin.

static load(*buff: bytes*) → Any

static load_dict(*representation: dict*) → Any

static name() → str
The name of the plugin.

plot(*plt: Any*, *X:* synthcity.plugins.core.dataloader.DataLoader, *count: Optional[int] = None*, *plots: list = ['marginal', 'associations', 'tsne']*, ***kwargs: Any*) → Any
Plot the real-synthetic distributions.

Parameters

- **plt** – output
- **X** – DataLoader. The reference dataset.

Returns self

classmethod sample_hyperparameters(**args: Any*, ***kwargs: Any*) → Dict[str, Any]
Sample value from the hyperparameter space for the current plugin.

classmethod sample_hyperparameters_optuna(*trial: Any*, **args: Any*, ***kwargs: Any*) → Dict[str, Any]

save() → bytes

save_dict() → dict

save_to_file(*path: pathlib.Path*) → bytes

schema() → *synthcity.plugins.core.schema.Schema*
The reference schema

schema_includes(*other: Union[*synthcity.plugins.core.dataloader.DataLoader, *pandas.core.frame.DataFrame]*) → bool
Helper method to test if the reference schema includes a Dataset

Parameters **other** – DataLoader. The dataset to test

Returns bool, if the schema includes the dataset or not.

training_schema() → *synthcity.plugins.core.schema.Schema*
The internal schema

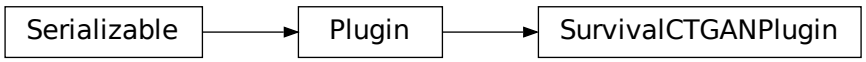
static type() → str
The type of the plugin.

static version() → str
API version

plugin
alias of *synthcity.plugins.survival_analysis.plugin_survival_gan.SurvivalGANPlugin*

synthcity.plugins.survival_analysis.plugin_survival_ctgan module

class SurvivalCTGANPlugin(*uncensoring_model: str = 'survival_function_regression'*, *dataloader_sampling_strategy: str = 'imbalanced_time_censoring'*, *tte_strategy: str = 'survival_function'*, *censoring_strategy: str = 'random'*, *device: Any = device(type='cpu')*, *workspace: pathlib.Path = PosixPath('workspace')*, *random_state: int = 0*, *compress_dataset: bool = False*, *sampling_patience: int = 500*, ***kwargs: Any*)
Bases: *synthcity.plugins.core.plugin.Plugin*



Survival Analysis Pipeline based on Conditional Tabular GANs.

Parameters

- **uncensoring_model** – str The time-to-event model: “survival_function_regression”.
- **dataloader_sampling_strategy** – str, default = imbalanced_time_censoring Training sampling strategy: none, imbalanced_censoring, imbalanced_time_censoring
- **tte_strategy** – str The time-to-event generation strategy: survival_function, uncensoring.
- **censoring_strategy** – str For the generated data, how to censor subjects: “random” or “covariate_dependent”
- **device** – torch device to use for training(cpu/cuda)
- **kwargs** – Any “ctgan” additional args, like n_iter = 100 etc.
- **arguments** (*# Core Plugin*) –
- **workspace** – Path. Optional Path for caching intermediary results.
- **compress_dataset** – bool. Default = False. Drop redundant features before training the generator.
- **sampling_patience** – int. Max inference iterations to wait for the generated data to match the training schema.

Example

```
>>> from lifelines.datasets import load_rossi
>>> from synthcity.plugins import Plugins
>>> from synthcity.plugins.core.dataloader import SurvivalAnalysisDataLoader
>>>
>>> X = load_rossi()
>>> data = SurvivalAnalysisDataLoader(
>>>        X,
>>>        target_column="arrest",
>>>        time_to_event_column="week",
>>> )
>>>
>>> plugin = Plugins().get("survival_ctgan")
>>> plugin.fit(data)
>>>
>>> plugin.generate(count = 50)
```

class Config

Bases: object

arbitrary_types_allowed = True

validate_assignment = True

fit(*X: Union[*synthcity.plugins.core.dataloader.DataLoader*, pandas.core.frame.DataFrame]*, **args: Any*, ***kwargs: Any*) → Any

Training method the synthetic data plugin.

Parameters

- **X** – DataLoader. The reference dataset.
- **cond** – Optional, Union[pd.DataFrame, pd.Series, np.ndarray] Optional Training Conditional. The training conditional can be used to control to output of some models, like GANs or VAEs. The content can be anything, as long as it maps to the training dataset X. Usage example:

```
>>> from sklearn.datasets import load_iris
>>> from synthcity.plugins.core.dataloader import␣
↪GenericDataLoader
>>> from synthcity.plugins.core.constraints import Constraints
>>>
>>> # Load in `test_plugin` the generative model of choice
>>> # ....
>>>
>>> X, y = load_iris(as_frame=True, return_X_y=True)
>>> X["target"] = y
>>>
>>> X = GenericDataLoader(X)
>>> test_plugin.fit(X, cond=y)
>>>
>>> count = 10
>>> X_gen = test_plugin.generate(count, cond=np.ones(count))
>>>
>>> # The Conditional only optimizes the output generation
```

(continues on next page)

(continued from previous page)

```
>>> # for GANs and VAEs, but does NOT guarantee the samples
>>> # are only from that condition.
>>> # If you want to guarantee that output contains only
>>> # "target" == 1 samples, use Constraints.
>>>
>>> constraints = Constraints(
>>>     rules=[
>>>         ("target", "==", 1),
>>>     ]
>>> )
>>> X_gen = test_plugin.generate(count,
>>>         cond=np.ones(count),
>>>         constraints=constraints
>>>        )
>>> assert (X_gen["target"] == 1).all()
```

Returns self

classmethod fqdn() → str
The Fully-Qualified name of the plugin.

generate(*count: Optional[int] = None*, *constraints: Optional[*synthcity.plugins.core.constraints.Constraints*] = None*, *random_state: Optional[int] = None*, ***kwargs: Any*) → *synthcity.plugins.core.dataloader.DataLoader*
Synthetic data generation method.

Parameters

- **count** – optional int. The number of samples to generate. If None, it generated len(reference_dataset) samples.
- **cond** – Optional, Union[pd.DataFrame, pd.Series, np.ndarray]. Optional Generation Conditional. The conditional can be used only if the model was trained using a conditional too. If provided, it must have *count* length. Not all models support conditionals. The conditionals can be used in VAEs or GANs to speed-up the generation under some constraints. For model agnostic solutions, check out the *constraints* parameter.
- **constraints** – optional Constraints. Optional constraints to apply on the generated data. If none, the reference schema constraints are applied. The constraints are model agnostic, and will filter the output of the generative model. The constraints are a list of rules. Each rule is a tuple of the form (<feature>, <operation>, <value>).

  Valid Operations:

  - ”<”, “lt” : less than <value>
  - ”<=”, “le”: less or equal with <value>
  - ”>”, “gt” : greater than <value>
  - ”>=”, “ge”: greater or equal with <value>
  - ”==”, “eq”: equal with <value>
  - ”in”: valid for categorical features, and <value> must be array. for example, (“target”, “in”, [0, 1])
  - ”dtype”: <value> can be a data type. For example, (“target”, “dtype”, “int”)

  Usage example:

```
>>> from synthcity.plugins.core.constraints import Constraints
>>> constraints = Constraints(
>>>   rules=[
>>>             ("InterestingFeature", "==", 0),
>>>         ]
>>>     )
>>>
>>> syn_data = syn_model.generate(
        count=count,
        constraints=constraints
    ).dataframe()
>>>
>>> assert (syn_data["InterestingFeature"] == 0).all()
```

- **random_state** – optional int. Optional random seed to use.

Returns <count> synthetic samples

static hyperparameter_space(***kwargs: Any*) → List[*synthcity.plugins.core.distribution.Distribution*]
Returns the hyperparameter space for the derived plugin.

static load(*buff: bytes*) → Any

static load_dict(*representation: dict*) → Any

static name() → str
The name of the plugin.

plot(*plt: Any*, *X:* synthcity.plugins.core.dataloader.DataLoader, *count: Optional[int] = None*, *plots: list = ['marginal', 'associations', 'tsne']*, ***kwargs: Any*) → Any
Plot the real-synthetic distributions.

Parameters

- **plt** – output
- **X** – DataLoader. The reference dataset.

Returns self

classmethod sample_hyperparameters(**args: Any*, ***kwargs: Any*) → Dict[str, Any]
Sample value from the hyperparameter space for the current plugin.

classmethod sample_hyperparameters_optuna(*trial: Any*, **args: Any*, ***kwargs: Any*) → Dict[str, Any]

save() → bytes

save_dict() → dict

save_to_file(*path: pathlib.Path*) → bytes

schema() → *synthcity.plugins.core.schema.Schema*
The reference schema

schema_includes(*other: Union[*synthcity.plugins.core.dataloader.DataLoader, *pandas.core.frame.DataFrame]*) → bool
Helper method to test if the reference schema includes a Dataset

Parameters **other** – DataLoader. The dataset to test

Returns bool, if the schema includes the dataset or not.

training_schema() → *synthcity.plugins.core.schema.Schema*
The internal schema

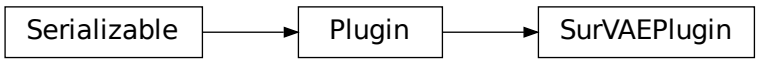
static type() → str
The type of the plugin.

static version() → str
API version

plugin
alias of *synthcity.plugins.survival_analysis.plugin_survival_ctgan.SurvivalCTGANPlugin*

synthcity.plugins.survival_analysis.plugin_survae module

class SurVAEPlugin(*uncensoring_model: str = 'survival_function_regression'*, *dataloader_sampling_strategy: str = 'imbalanced_time_censoring'*, *tte_strategy: str = 'survival_function'*, *censoring_strategy: str = 'random'*, *device: Any = device(type='cpu')*, *workspace: pathlib.Path = PosixPath('workspace')*, *random_state: int = 0*, *compress_dataset: bool = False*, *sampling_patience: int = 500*, ***kwargs: Any*)
Bases: *synthcity.plugins.core.plugin.Plugin*



Survival Analysis Pipeline based on Variational AutoEncoders.

Parameters

- **uncensoring_model** –

  str The time-to-event model: “survival_function_regression”.

  dataloader_sampling_strategy: str, default = imbalanced_time_censoring Training sampling strategy: none, imbalanced_censoring, imbalanced_time_censoring

  tte_strategy: str

  The time-to-event generation strategy: survival_function, uncensoring.

  censoring_strategy: str For the generated data, how to censor subjects: “random” or “covariate_dependent”

- **device** – torch device to use for training(cpu/cuda) kwargs: Any

  ”tvae” additional args, like n_iter = 100 etc.

  # Core Plugin arguments workspace: Path.

  Optional Path for caching intermediary results.

  compress_dataset: bool. Default = False. Drop redundant features before training the generator.

sampling_patience: int. Max inference iterations to wait for the generated data to match the training schema.

Example

```
>>> from lifelines.datasets import load_rossi
>>> from synthcity.plugins import Plugins
>>> from synthcity.plugins.core.dataloader import SurvivalAnalysisDataLoader
>>>
>>> X = load_rossi()
>>> data = SurvivalAnalysisDataLoader(
>>>        X,
>>>        target_column="arrest",
>>>        time_to_event_column="week",
>>> )
>>>
>>> plugin = Plugins().get("survae")
>>> plugin.fit(data)
>>>
>>> plugin.generate(count = 50)
```

class Config

Bases: object

arbitrary_types_allowed = True

validate_assignment = True

fit(*X: Union[*synthcity.plugins.core.dataloader.DataLoader*, pandas.core.frame.DataFrame]*, **args: Any*, ***kwargs: Any*) → Any

Training method the synthetic data plugin.

Parameters

- **X** – DataLoader. The reference dataset.
- **cond** – Optional, Union[pd.DataFrame, pd.Series, np.ndarray] Optional Training Conditional. The training conditional can be used to control to output of some models, like GANs or VAEs. The content can be anything, as long as it maps to the training dataset X. Usage example:

```
>>> from sklearn.datasets import load_iris
>>> from synthcity.plugins.core.dataloader import␣
↪GenericDataLoader
>>> from synthcity.plugins.core.constraints import Constraints
>>>
>>> # Load in `test_plugin` the generative model of choice
>>> # ....
>>>
>>> X, y = load_iris(as_frame=True, return_X_y=True)
>>> X["target"] = y
>>>
>>> X = GenericDataLoader(X)
>>> test_plugin.fit(X, cond=y)
>>>
```

(continues on next page)

(continued from previous page)

```
>>> count = 10
>>> X_gen = test_plugin.generate(count, cond=np.ones(count))
>>>
>>> # The Conditional only optimizes the output generation
>>> # for GANs and VAEs, but does NOT guarantee the samples
>>> # are only from that condition.
>>> # If you want to guarantee that output contains only
>>> # "target" == 1 samples, use Constraints.
>>>
>>> constraints = Constraints(
>>>     rules=[
>>>         ("target", "==", 1),
>>>     ]
>>> )
>>> X_gen = test_plugin.generate(count,
>>>         cond=np.ones(count),
>>>         constraints=constraints
>>>        )
>>> assert (X_gen["target"] == 1).all()
```

Returns self

classmethod fqdn() → str
The Fully-Qualified name of the plugin.

generate(*count: Optional[int] = None*, *constraints: Optional[*synthcity.plugins.core.constraints.Constraints*] = None*, *random_state: Optional[int] = None*, ***kwargs: Any*) → *synthcity.plugins.core.dataloader.DataLoader*
Synthetic data generation method.

Parameters

- **count** – optional int. The number of samples to generate. If None, it generated len(reference_dataset) samples.
- **cond** – Optional, Union[pd.DataFrame, pd.Series, np.ndarray]. Optional Generation Conditional. The conditional can be used only if the model was trained using a conditional too. If provided, it must have *count* length. Not all models support conditionals. The conditionals can be used in VAEs or GANs to speed-up the generation under some constraints. For model agnostic solutions, check out the *constraints* parameter.
- **constraints** – optional Constraints. Optional constraints to apply on the generated data. If none, the reference schema constraints are applied. The constraints are model agnostic, and will filter the output of the generative model. The constraints are a list of rules. Each rule is a tuple of the form (<feature>, <operation>, <value>).

  Valid Operations:

  - ”<”, “lt” : less than <value>
  - ”<=”, “le”: less or equal with <value>
  - ”>”, “gt” : greater than <value>
  - ”>=”, “ge”: greater or equal with <value>
  - ”==”, “eq”: equal with <value>

- ”in”: valid for categorical features, and <value> must be array. for example, (“target”, “in”, [0, 1])
- ”dtype”: <value> can be a data type. For example, (“target”, “dtype”, “int”)

Usage example:

```
>>> from synthcity.plugins.core.constraints import Constraints
>>> constraints = Constraints(
>>>   rules=[
>>>             ("InterestingFeature", "==", 0),
>>>         ]
>>>     )
>>>
>>> syn_data = syn_model.generate(
        count=count,
        constraints=constraints
    ).dataframe()
>>>
>>> assert (syn_data["InterestingFeature"] == 0).all()
```

- **random_state** – optional int. Optional random seed to use.

Returns <count> synthetic samples

static hyperparameter_space(***kwargs: Any*) → List[*synthcity.plugins.core.distribution.Distribution*]
Returns the hyperparameter space for the derived plugin.

static load(*buff: bytes*) → Any

static load_dict(*representation: dict*) → Any

static name() → str
The name of the plugin.

plot(*plt: Any*, *X:* synthcity.plugins.core.dataloader.DataLoader, *count: Optional[int] = None*, *plots: list = ['marginal', 'associations', 'tsne']*, ***kwargs: Any*) → Any
Plot the real-synthetic distributions.

Parameters

- **plt** – output
- **X** – DataLoader. The reference dataset.

Returns self

classmethod sample_hyperparameters(**args: Any*, ***kwargs: Any*) → Dict[str, Any]
Sample value from the hyperparameter space for the current plugin.

classmethod sample_hyperparameters_optuna(*trial: Any*, **args: Any*, ***kwargs: Any*) → Dict[str, Any]

save() → bytes

save_dict() → dict

save_to_file(*path: pathlib.Path*) → bytes

schema() → *synthcity.plugins.core.schema.Schema*
The reference schema

schema_includes(*other: Union[synthcity.plugins.core.dataloader.DataLoader, pandas.core.frame.DataFrame]*) → bool
Helper method to test if the reference schema includes a Dataset

Parameters **other** – DataLoader. The dataset to test

Returns bool, if the schema includes the dataset or not.

training_schema() → *synthcity.plugins.core.schema.Schema*
The internal schema

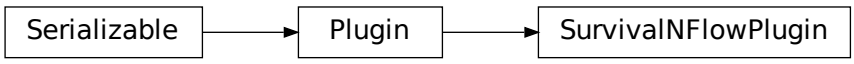
static type() → str
The type of the plugin.

static version() → str
API version

plugin
alias of *synthcity.plugins.survival_analysis.plugin_survae.SurVAEPlugin*

synthcity.plugins.survival_analysis.plugin_survival_nflow module

class SurvivalNFlowPlugin(*uncensoring_model: str = 'survival_function_regression', dataloader_sampling_strategy: str = 'imbalanced_time_censoring', tte_strategy: str = 'survival_function', censoring_strategy: str = 'random', device: Any = device(type='cpu'), workspace: pathlib.Path = PosixPath('workspace'), random_state: int = 0, compress_dataset: bool = False, sampling_patience: int = 500, **kwargs: Any*)
Bases: *synthcity.plugins.core.plugin.Plugin*



Survival Analysis Pipeline based on Normalizing flows.

Parameters **uncensoring_model** –

str The time-to-event model: “survival_function_regression”.

dataloader_sampling_strategy: str, default = imbalanced_time_censoring Training sampling strategy: none, imbalanced_censoring, imbalanced_time_censoring

tte_strategy: str

The time-to-event generation strategy: survival_function, uncensoring.

censoring_strategy: str For the generated data, how to censor subjects: “random” or “covariate_dependent”

kwargs: Any ”nflow” additional args, like n_iter = 100 etc.

device: torch device to use for training(cpu/cuda)

Core Plugin arguments workspace: Path.

Optional Path for caching intermediary results.

compress_dataset: bool. Default = False. Drop redundant features before training the generator.

sampling_patience: int. Max inference iterations to wait for the generated data to match the training schema.

Example

```
>>> from lifelines.datasets import load_rossi
>>> from synthcity.plugins import Plugins
>>> from synthcity.plugins.core.dataloader import SurvivalAnalysisDataLoader
>>>
>>> X = load_rossi()
>>> data = SurvivalAnalysisDataLoader(
>>>        X,
>>>        target_column="arrest",
>>>        time_to_event_column="week",
>>> )
>>>
>>> plugin = Plugins().get("survival_ctgan")
>>> plugin.fit(data)
>>>
>>> plugin.generate(count = 50)
```

class Config

Bases: object

arbitrary_types_allowed = True

validate_assignment = True

fit(*X: Union[*synthcity.plugins.core.dataloader.DataLoader*, pandas.core.frame.DataFrame]*, **args: Any*, ***kwargs: Any*) → Any

Training method the synthetic data plugin.

Parameters

- **X** – DataLoader. The reference dataset.
- **cond** – Optional, Union[pd.DataFrame, pd.Series, np.ndarray] Optional Training Conditional. The training conditional can be used to control to output of some models, like GANs or VAEs. The content can be anything, as long as it maps to the training dataset X. Usage example:

```
>>> from sklearn.datasets import load_iris
>>> from synthcity.plugins.core.dataloader import␣
↪GenericDataLoader
>>> from synthcity.plugins.core.constraints import Constraints
>>>
>>> # Load in `test_plugin` the generative model of choice
>>> # ....
>>>
>>> X, y = load_iris(as_frame=True, return_X_y=True)
>>> X["target"] = y
```

(continues on next page)

(continued from previous page)

```
>>>
>>> X = GenericDataLoader(X)
>>> test_plugin.fit(X, cond=y)
>>>
>>> count = 10
>>> X_gen = test_plugin.generate(count, cond=np.ones(count))
>>>
>>> # The Conditional only optimizes the output generation
>>> # for GANs and VAEs, but does NOT guarantee the samples
>>> # are only from that condition.
>>> # If you want to guarantee that output contains only
>>> # "target" == 1 samples, use Constraints.
>>>
>>> constraints = Constraints(
>>>     rules=[
>>>         ("target", "==", 1),
>>>     ]
>>> )
>>> X_gen = test_plugin.generate(count,
>>>         cond=np.ones(count),
>>>         constraints=constraints
>>>        )
>>> assert (X_gen["target"] == 1).all()
```

Returns self

classmethod fqdn() → str
The Fully-Qualified name of the plugin.

generate(*count: Optional[int] = None*, *constraints: Optional[*synthcity.plugins.core.constraints.Constraints*] = None*, *random_state: Optional[int] = None*, ***kwargs: Any*) → *synthcity.plugins.core.dataloader.DataLoader*
Synthetic data generation method.

Parameters

- **count** – optional int. The number of samples to generate. If None, it generated len(reference_dataset) samples.
- **cond** – Optional, Union[pd.DataFrame, pd.Series, np.ndarray]. Optional Generation Conditional. The conditional can be used only if the model was trained using a conditional too. If provided, it must have *count* length. Not all models support conditionals. The conditionals can be used in VAEs or GANs to speed-up the generation under some constraints. For model agnostic solutions, check out the *constraints* parameter.
- **constraints** – optional Constraints. Optional constraints to apply on the generated data. If none, the reference schema constraints are applied. The constraints are model agnostic, and will filter the output of the generative model. The constraints are a list of rules. Each rule is a tuple of the form (<feature>, <operation>, <value>).

  Valid Operations:

  - "<", "lt" : less than <value>
  - "<=", "le": less or equal with <value>
  - ">", "gt" : greater than <value>

- ">=", "ge": greater or equal with <value>
- "==", "eq": equal with <value>
- "in": valid for categorical features, and <value> must be array. for example, ("target", "in", [0, 1])
- "dtype": <value> can be a data type. For example, ("target", "dtype", "int")

Usage example:

```
>>> from synthcity.plugins.core.constraints import Constraints
>>> constraints = Constraints(
>>>   rules=[
>>>             ("InterestingFeature", "==", 0),
>>>         ]
>>>     )
>>>
>>> syn_data = syn_model.generate(
        count=count,
        constraints=constraints
    ).dataframe()
>>>
>>> assert (syn_data["InterestingFeature"] == 0).all()
```

- **random_state** – optional int. Optional random seed to use.

Returns <count> synthetic samples

static hyperparameter_space(***kwargs: Any*) → List[*synthcity.plugins.core.distribution.Distribution*]
Returns the hyperparameter space for the derived plugin.

static load(*buff: bytes*) → Any

static load_dict(*representation: dict*) → Any

static name() → str
The name of the plugin.

plot(*plt: Any*, *X:* synthcity.plugins.core.dataloader.DataLoader, *count: Optional[int] = None*, *plots: list = ['marginal', 'associations', 'tsne']*, ***kwargs: Any*) → Any
Plot the real-synthetic distributions.

Parameters

- **plt** – output
- **X** – DataLoader. The reference dataset.

Returns self

classmethod sample_hyperparameters(**args: Any*, ***kwargs: Any*) → Dict[str, Any]
Sample value from the hyperparameter space for the current plugin.

classmethod sample_hyperparameters_optuna(*trial: Any*, **args: Any*, ***kwargs: Any*) → Dict[str, Any]

save() → bytes

save_dict() → dict

save_to_file(*path: pathlib.Path*) → bytes

schema() → *synthcity.plugins.core.schema.Schema*
The reference schema

schema_includes(*other: Union[*synthcity.plugins.core.dataloader.DataLoader*, pandas.core.frame.DataFrame]*) → bool
Helper method to test if the reference schema includes a Dataset

Parameters **other** – DataLoader. The dataset to test

Returns bool, if the schema includes the dataset or not.

training_schema() → *synthcity.plugins.core.schema.Schema*
The internal schema

static type() → str
The type of the plugin.

static version() → str
API version

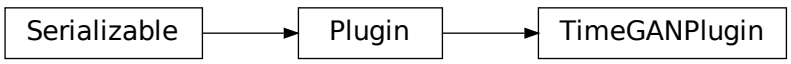
plugin
alias of *synthcity.plugins.survival_analysis.plugin_survival_nflow.SurvivalNFlowPlugin*

10.1.5 Time-series & Time-Series Survival Analysis

synthcity.plugins.time_series.plugin_timegan module

Reference: "Time-series Generative Adversarial Networks", Jinsung Yoon, Daniel Jarrett, Mihaela van der Schaar

class TimeGANPlugin(*n_iter: int = 1000*, *generator_n_layers_hidden: int = 2*, *generator_n_units_hidden: int = 150*, *generator_nonlin: str = 'leaky_relu'*, *generator_nonlin_out_discrete: str = 'softmax'*, *generator_nonlin_out_continuous: str = 'tanh'*, *generator_batch_norm: bool = False*, *generator_dropout: float = 0.01*, *generator_loss: Optional[Callable] = None*, *generator_lr: float = 0.001*, *generator_weight_decay: float = 0.001*, *generator_residual: bool = True*, *discriminator_n_layers_hidden: int = 3*, *discriminator_n_units_hidden: int = 300*, *discriminator_nonlin: str = 'leaky_relu'*, *discriminator_n_iter: int = 1*, *discriminator_batch_norm: bool = False*, *discriminator_dropout: float = 0.1*, *discriminator_loss: Optional[Callable] = None*, *discriminator_lr: float = 0.001*, *discriminator_weight_decay: float = 0.001*, *batch_size: int = 64*, *n_iter_print: int = 10*, *clipping_value: int = 0*, *encoder_max_clusters: int = 20*, *encoder: Optional[Any] = None*, *device: Any = device(type='cpu')*, *mode: str = 'RNN'*, *gamma_penalty: float = 1*, *moments_penalty: float = 100*, *embedding_penalty: float = 10*, *use_horizon_condition: bool = True*, *dataloader_sampling_strategy: str = 'imbalanced_time_censoring'*, *random_state: int = 0*, *workspace: pathlib.Path = PosixPath('workspace')*, *compress_dataset: bool = False*, *sampling_patience: int = 500*, ***kwargs: Any*)
Bases: *synthcity.plugins.core.plugin.Plugin*



Synthetic time series generation using TimeGAN.

Parameters

- **n_iter** – int Maximum number of iterations in the Generator.
- **n_units_in** – int Number of features
- **generator_n_layers_hidden** – int Number of hidden layers in the generator
- **generator_n_units_hidden** – int Number of hidden units in each layer of the Generator
- **generator_nonlin** – string, default 'elu' Nonlinearity to use in the generator. Can be 'elu', 'relu', 'selu' or 'leaky_relu'.
- **generator_batch_norm** – bool Enable/disable batch norm for the generator
- **generator_dropout** – float Dropout value. If 0, the dropout is not used.
- **generator_residual** – bool Use residuals for the generator
- **discriminator_n_layers_hidden** – int Number of hidden layers in the discriminator
- **discriminator_n_units_hidden** – int Number of hidden units in each layer of the discriminator
- **discriminator_nonlin** – string, default 'relu' Nonlinearity to use in the discriminator. Can be 'elu', 'relu', 'selu' or 'leaky_relu'.
- **discriminator_n_iter** – int Maximum number of iterations in the discriminator.
- **discriminator_batch_norm** – bool Enable/disable batch norm for the discriminator
- **discriminator_dropout** – float Dropout value for the discriminator. If 0, the dropout is not used.
- **lr** – float learning rate for optimizer.
- **weight_decay** – float l2 (ridge) penalty for the weights.
- **batch_size** – int Batch size
- **n_iter_print** – int Number of iterations after which to print updates and check the validation loss.
- **random_state** – int random_state used
- **clipping_value** – int, default 0 Gradients clipping value
- **gamma_penalty** – Latent representation penalty
- **moments_penalty** – float = 100 Moments(var and mean) penalty
- **embedding_penalty** – float = 10 Embedding representation penalty
- **mode** – str = "RNN" Core neural net architecture. Available models:
  - "LSTM"
  - "GRU"
  - "RNN"
  - "Transformer"
  - "MLSTM_FCN"
  - "TCN"
  - "InceptionTime"

  - "InceptionTimePlus"
  - "XceptionTime"
  - "ResCNN"
  - "OmniScaleCNN"
  - "XCM"
- **device** – The device used by PyTorch. cpu/cuda
- **use_horizon_condition** – bool. Default = True Whether to condition the covariate generation on the observation times or not.
- **encoder_max_clusters** – int The max number of clusters to create for continuous columns when encoding
- **encoder** – Pre-trained tabular encoder. If None, a new encoder is trained.
- **arguments** (*# Core Plugin*) –
- **workspace** – Path. Optional Path for caching intermediary results.
- **compress_dataset** – bool. Default = False. Drop redundant features before training the generator.
- **sampling_patience** – int. Max inference iterations to wait for the generated data to match the training schema.

Example

```
>>> from synthcity.plugins import Plugins
>>> from synthcity.utils.datasets.time_series.google_stocks import␣
↪GoogleStocksDataloader
>>> from synthcity.plugins.core.dataloader import TimeSeriesDataLoader
>>> static, temporal, horizons, outcome = GoogleStocksDataloader().load()
>>> loader = TimeSeriesDataLoader(
>>>             temporal_data=temporal,
>>>             observation_times=horizons,
>>>             static_data=static,
>>>             outcome=outcome,
>>> )
>>>
>>> plugin = Plugins().get("timegan", n_iter = 50)
>>> plugin.fit(loader)
>>>
>>> plugin.generate(count = 10)
```

class Config

Bases: object

arbitrary_types_allowed = True

validate_assignment = True

fit(*X: Union[*synthcity.plugins.core.dataloader.DataLoader*, pandas.core.frame.DataFrame]*, **args: Any*, ***kwargs: Any*) → Any

Training method the synthetic data plugin.

Parameters

- **X** – DataLoader. The reference dataset.
- **cond** – Optional, Union[pd.DataFrame, pd.Series, np.ndarray] Optional Training Conditional. The training conditional can be used to control to output of some models, like GANs or VAEs. The content can be anything, as long as it maps to the training dataset X. Usage example:

```
>>> from sklearn.datasets import load_iris
>>> from synthcity.plugins.core.dataloader import␣
↪GenericDataLoader
>>> from synthcity.plugins.core.constraints import Constraints
>>>
>>> # Load in `test_plugin` the generative model of choice
>>> # ....
>>>
>>> X, y = load_iris(as_frame=True, return_X_y=True)
>>> X["target"] = y
>>>
>>> X = GenericDataLoader(X)
>>> test_plugin.fit(X, cond=y)
>>>
>>> count = 10
>>> X_gen = test_plugin.generate(count, cond=np.ones(count))
>>>
>>> # The Conditional only optimizes the output generation
>>> # for GANs and VAEs, but does NOT guarantee the samples
>>> # are only from that condition.
>>> # If you want to guarantee that output contains only
>>> # "target" == 1 samples, use Constraints.
>>>
>>> constraints = Constraints(
>>>     rules=[
>>>         ("target", "==", 1),
>>>     ]
>>> )
>>> X_gen = test_plugin.generate(count,
>>>         cond=np.ones(count),
>>>         constraints=constraints
>>>        )
>>> assert (X_gen["target"] == 1).all()
```

Returns self

classmethod fqdn() → str
The Fully-Qualified name of the plugin.

generate(*count: Optional[int] = None*, *constraints: Optional[*synthcity.plugins.core.constraints.Constraints*] = None*, *random_state: Optional[int] = None*, ***kwargs: Any*) → *synthcity.plugins.core.dataloader.DataLoader*
Synthetic data generation method.

Parameters

- **count** – optional int. The number of samples to generate. If None, it generated len(reference_dataset) samples.
- **cond** – Optional, Union[pd.DataFrame, pd.Series, np.ndarray]. Optional Generation

Conditional. The conditional can be used only if the model was trained using a conditional too. If provided, it must have *count* length. Not all models support conditionals. The conditionals can be used in VAEs or GANs to speed-up the generation under some constraints. For model agnostic solutions, check out the *constraints* parameter.

- **constraints** – optional Constraints. Optional constraints to apply on the generated data. If none, the reference schema constraints are applied. The constraints are model agnostic, and will filter the output of the generative model. The constraints are a list of rules. Each rule is a tuple of the form (<feature>, <operation>, <value>).

  Valid Operations:

  - "<", "lt" : less than <value>
  - "<=", "le": less or equal with <value>
  - ">", "gt" : greater than <value>
  - ">=", "ge": greater or equal with <value>
  - "==", "eq": equal with <value>
  - "in": valid for categorical features, and <value> must be array. for example, ("target", "in", [0, 1])
  - "dtype": <value> can be a data type. For example, ("target", "dtype", "int")

  Usage example:

```
>>> from synthcity.plugins.core.constraints import Constraints
>>> constraints = Constraints(
>>>   rules=[
>>>             ("InterestingFeature", "==", 0),
>>>         ]
>>>     )
>>>
>>> syn_data = syn_model.generate(
        count=count,
        constraints=constraints
    ).dataframe()
>>>
>>> assert (syn_data["InterestingFeature"] == 0).all()
```

- **random_state** – optional int. Optional random seed to use.

Returns <count> synthetic samples

static hyperparameter_space(***kwargs: Any*) → List[*synthcity.plugins.core.distribution.Distribution*]
Returns the hyperparameter space for the derived plugin.

static load(*buff: bytes*) → Any

static load_dict(*representation: dict*) → Any

static name() → str
The name of the plugin.

plot(*plt: Any*, *X:* synthcity.plugins.core.dataloader.DataLoader, *count: Optional[int] = None*, *plots: list = ['marginal', 'associations', 'tsne']*, ***kwargs: Any*) → Any
Plot the real-synthetic distributions.

Parameters

- **plt** – output
- **X** – DataLoader. The reference dataset.

Returns self

classmethod sample_hyperparameters(**args: Any*, ***kwargs: Any*) → Dict[str, Any]
Sample value from the hyperparameter space for the current plugin.

classmethod sample_hyperparameters_optuna(*trial: Any*, **args: Any*, ***kwargs: Any*) → Dict[str, Any]

save() → bytes

save_dict() → dict

save_to_file(*path: pathlib.Path*) → bytes

schema() → *synthcity.plugins.core.schema.Schema*
The reference schema

schema_includes(*other: Union[*synthcity.plugins.core.dataloader.DataLoader*, pandas.core.frame.DataFrame]*) → bool
Helper method to test if the reference schema includes a Dataset

Parameters **other** – DataLoader. The dataset to test

Returns bool, if the schema includes the dataset or not.

training_schema() → *synthcity.plugins.core.schema.Schema*
The internal schema

static type() → str
The type of the plugin.

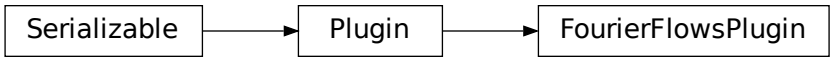
static version() → str
API version

plugin
alias of *synthcity.plugins.time_series.plugin_timegan.TimeGANPlugin*

synthcity.plugins.time_series.plugin_fflows module

Reference: "Generative Time-series Modeling with Fourier Flows", Ahmed Alaa, Alex Chan, and Mihaela van der Schaar.

class FourierFlowsPlugin(*n_iter: int = 500*, *batch_size: int = 128*, *lr: float = 0.001*, *n_iter_print: int = 100*, *n_units_hidden: int = 100*, *n_flows: int = 10*, *FFT: bool = True*, *flip: bool = True*, *normalize: bool = False*, *static_model: str = 'ctgan'*, *device: Any = device(type='cpu')*, *encoder_max_clusters: int = 10*, *random_state: int = 0*, *workspace: pathlib.Path = PosixPath('workspace')*, *compress_dataset: bool = False*, *sampling_patience: int = 500*, ***kwargs: Any*)
Bases: *synthcity.plugins.core.plugin.Plugin*



Synthetic time series generation using FourierFlows.

Parameters

- **n_iter** – int Number of training iterations
- **batch_size** – int Batch size
- **lr** – float Learning rate
- **n_iter_print** – int Number of iterations to print the validation loss
- **n_units_hidden** – int Number of hidden nodes
- **n_flows** – int Number of flows to use(default = 10)
- **FFT** – bool Use Fourier transform(default = True)
- **flip** – bool Flip the data in the SpectralFilter
- **normalize** – bool Scale the data(default = False)
- **static_model** – str = "ctgan", The model to use for generating the static data.
- **device** – Any = DEVICE torch device to use for training(cpu/cuda)
- **encoder_max_clusters** – int = 10 Number of clusters used for tabular encoding
- **arguments** (*# Core Plugin*) –
- **workspace** – Path. Optional Path for caching intermediary results.
- **compress_dataset** – bool. Default = False. Drop redundant features before training the generator.
- **sampling_patience** – int. Max inference iterations to wait for the generated data to match the training schema.

Example

```
>>> from synthcity.plugins import Plugins
>>> from synthcity.utils.datasets.time_series.google_stocks import␣
↪GoogleStocksDataloader
>>> from synthcity.plugins.core.dataloader import TimeSeriesDataLoader
>>> static, temporal, horizons, outcome = GoogleStocksDataloader().load()
>>> loader = TimeSeriesDataLoader(
>>>             temporal_data=temporal,
>>>             observation_times=horizons,
>>>             static_data=static,
>>>             outcome=outcome,
>>> )
>>>
```

(continues on next page)

(continued from previous page)

```
>>> plugin = Plugins().get("fflows", n_iter = 50)
>>> plugin.fit(loader)
>>>
>>> plugin.generate(count = 10)
```

class Config

Bases: object

arbitrary_types_allowed = True

validate_assignment = True

fit(*X: Union[*synthcity.plugins.core.dataloader.DataLoader*, pandas.core.frame.DataFrame]*, **args: Any*, ***kwargs: Any*) → Any

Training method the synthetic data plugin.

Parameters

- **X** – DataLoader. The reference dataset.
- **cond** – Optional, Union[pd.DataFrame, pd.Series, np.ndarray] Optional Training Conditional. The training conditional can be used to control to output of some models, like GANs or VAEs. The content can be anything, as long as it maps to the training dataset X. Usage example:

```
>>> from sklearn.datasets import load_iris
>>> from synthcity.plugins.core.dataloader import␣
↪GenericDataLoader
>>> from synthcity.plugins.core.constraints import Constraints
>>>
>>> # Load in `test_plugin` the generative model of choice
>>> # ....
>>>
>>> X, y = load_iris(as_frame=True, return_X_y=True)
>>> X["target"] = y
>>>
>>> X = GenericDataLoader(X)
>>> test_plugin.fit(X, cond=y)
>>>
>>> count = 10
>>> X_gen = test_plugin.generate(count, cond=np.ones(count))
>>>
>>> # The Conditional only optimizes the output generation
>>> # for GANs and VAEs, but does NOT guarantee the samples
>>> # are only from that condition.
>>> # If you want to guarantee that output contains only
>>> # "target" == 1 samples, use Constraints.
>>>
>>> constraints = Constraints(
>>>     rules=[
>>>         ("target", "==", 1),
>>>     ]
>>> )
>>> X_gen = test_plugin.generate(count,
>>>         cond=np.ones(count),
```

(continues on next page)

(continued from previous page)

```
>>>         constraints=constraints
>>>        )
>>> assert (X_gen["target"] == 1).all()
```

Returns self

classmethod fqdn() → str
The Fully-Qualified name of the plugin.

generate(*count: Optional[int] = None*, *constraints: Optional[*synthcity.plugins.core.constraints.Constraints*] = None*, *random_state: Optional[int] = None*, ***kwargs: Any*) → *synthcity.plugins.core.dataloader.DataLoader*
Synthetic data generation method.

Parameters

- **count** – optional int. The number of samples to generate. If None, it generated len(reference_dataset) samples.
- **cond** – Optional, Union[pd.DataFrame, pd.Series, np.ndarray]. Optional Generation Conditional. The conditional can be used only if the model was trained using a conditional too. If provided, it must have *count* length. Not all models support conditionals. The conditionals can be used in VAEs or GANs to speed-up the generation under some constraints. For model agnostic solutions, check out the *constraints* parameter.
- **constraints** – optional Constraints. Optional constraints to apply on the generated data. If none, the reference schema constraints are applied. The constraints are model agnostic, and will filter the output of the generative model. The constraints are a list of rules. Each rule is a tuple of the form (<feature>, <operation>, <value>).

  Valid Operations:

  - "<", "lt" : less than <value>
  - "<=", "le": less or equal with <value>
  - ">", "gt" : greater than <value>
  - ">=", "ge": greater or equal with <value>
  - "==", "eq": equal with <value>
  - "in": valid for categorical features, and <value> must be array. for example, ("target", "in", [0, 1])
  - "dtype": <value> can be a data type. For example, ("target", "dtype", "int")

  Usage example:

```
>>> from synthcity.plugins.core.constraints import Constraints
>>> constraints = Constraints(
>>>   rules=[
>>>             ("InterestingFeature", "==", 0),
>>>         ]
>>>     )
>>>
>>> syn_data = syn_model.generate(
        count=count,
        constraints=constraints
    ).dataframe()
```

(continues on next page)

(continued from previous page)

```
>>>
>>> assert (syn_data["InterestingFeature"] == 0).all()
```

- **random_state** – optional int. Optional random seed to use.

Returns <count> synthetic samples

static hyperparameter_space(***kwargs: Any*) → List[*synthcity.plugins.core.distribution.Distribution*]
Returns the hyperparameter space for the derived plugin.

static load(*buff: bytes*) → Any

static load_dict(*representation: dict*) → Any

static name() → str
The name of the plugin.

plot(*plt: Any*, *X:* synthcity.plugins.core.dataloader.DataLoader, *count: Optional[int] = None*, *plots: list = ['marginal', 'associations', 'tsne']*, ***kwargs: Any*) → Any
Plot the real-synthetic distributions.

Parameters

- **plt** – output
- **X** – DataLoader. The reference dataset.

Returns self

classmethod sample_hyperparameters(**args: Any*, ***kwargs: Any*) → Dict[str, Any]
Sample value from the hyperparameter space for the current plugin.

classmethod sample_hyperparameters_optuna(*trial: Any*, **args: Any*, ***kwargs: Any*) → Dict[str, Any]

save() → bytes

save_dict() → dict

save_to_file(*path: pathlib.Path*) → bytes

schema() → *synthcity.plugins.core.schema.Schema*
The reference schema

schema_includes(*other: Union[*synthcity.plugins.core.dataloader.DataLoader*, pandas.core.frame.DataFrame]*) → bool
Helper method to test if the reference schema includes a Dataset

Parameters **other** – DataLoader. The dataset to test

Returns bool, if the schema includes the dataset or not.

training_schema() → *synthcity.plugins.core.schema.Schema*
The internal schema

static type() → str
The type of the plugin.

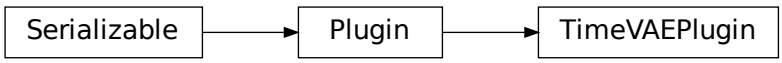
static version() → str
API version

plugin
alias of *synthcity.plugins.time_series.plugin_fflows.FourierFlowsPlugin*

synthcity.plugins.time_series.plugin_timevae module

class TimeVAEPlugin(*n_iter: int = 1000*, *decoder_n_layers_hidden: int = 2*, *decoder_n_units_hidden: int = 150*, *decoder_nonlin: str = 'leaky_relu'*, *decoder_nonlin_out_discrete: str = 'softmax'*, *decoder_nonlin_out_continuous: str = 'tanh'*, *decoder_batch_norm: bool = False*, *decoder_dropout: float = 0.01*, *decoder_residual: bool = True*, *encoder_n_layers_hidden: int = 3*, *encoder_n_units_hidden: int = 300*, *encoder_nonlin: str = 'leaky_relu'*, *encoder_batch_norm: bool = False*, *encoder_dropout: float = 0.1*, *lr: float = 0.001*, *weight_decay: float = 0.001*, *batch_size: int = 64*, *n_iter_print: int = 10*, *clipping_value: int = 0*, *encoder_max_clusters: int = 20*, *encoder: Optional[Any] = None*, *device: Any = device(type='cpu')*, *mode: str = 'LSTM'*, *gamma_penalty: float = 1*, *moments_penalty: float = 100*, *embedding_penalty: float = 10*, *random_state: int = 0*, *workspace: pathlib.Path = PosixPath('workspace')*, *compress_dataset: bool = False*, *sampling_patience: int = 500*, ***kwargs: Any*)

Bases: *synthcity.plugins.core.plugin.Plugin*



Synthetic time series generation using a Variational AutoEncoder.

Parameters

- **n_iter** – int Maximum number of iterations in the decoder.
- **n_units_in** – int Number of features
- **decoder_n_layers_hidden** – int Number of hidden layers in the decoder
- **decoder_n_units_hidden** – int Number of hidden units in each layer of the decoder
- **decoder_nonlin** – string, default ‘elu’ Nonlinearity to use in the decoder. Can be ‘elu’, ‘relu’, ‘selu’ or ‘leaky_relu’.
- **decoder_batch_norm** – bool Enable/disable batch norm for the decoder
- **decoder_dropout** – float Dropout value. If 0, the dropout is not used.
- **decoder_residual** – bool Use residuals for the decoder
- **encoder_n_layers_hidden** – int Number of hidden layers in the encoder
- **encoder_n_units_hidden** – int Number of hidden units in each layer of the encoder
- **encoder_nonlin** – string, default ‘relu’ Nonlinearity to use in the encoder. Can be ‘elu’, ‘relu’, ‘selu’ or ‘leaky_relu’.
- **encoder_n_iter** – int Maximum number of iterations in the encoder.
- **encoder_batch_norm** – bool Enable/disable batch norm for the encoder
- **encoder_dropout** – float Dropout value for the encoder. If 0, the dropout is not used.
- **lr** – float learning rate for optimizer.
- **weight_decay** – float l2 (ridge) penalty for the weights.

- **batch_size** – int Batch size
- **n_iter_print** – int Number of iterations after which to print updates and check the validation loss.
- **random_state** – int random_state used
- **clipping_value** – int, default 0 Gradients clipping value
- **mode** – str = "RNN" Core neural net architecture. Available models:
  - "LSTM"
  - "GRU"
  - "RNN"
  - "Transformer"
  - "MLSTM_FCN"
  - "TCN"
  - "InceptionTime"
  - "InceptionTimePlus"
  - "XceptionTime"
  - "ResCNN"
  - "OmniScaleCNN"
  - "XCM"
- **device** – The device used by PyTorch. cpu/cuda
- **use_horizon_condition** – bool. Default = True Whether to condition the covariate generation on the observation times or not.
- **encoder_max_clusters** – int The max number of clusters to create for continuous columns when encoding
- **encoder** – Pre-trained tabular encoder. If None, a new encoder is trained.
- **arguments** (*# Core Plugin*) –
- **workspace** – Path. Optional Path for caching intermediary results.
- **compress_dataset** – bool. Default = False. Drop redundant features before training the generator.
- **sampling_patience** – int. Max inference iterations to wait for the generated data to match the training schema.

Example

```
>>> from synthcity.plugins import Plugins
>>> from synthcity.utils.datasets.time_series.google_stocks import␣
↪GoogleStocksDataloader
>>> from synthcity.plugins.core.dataloader import TimeSeriesDataLoader
>>>
>>> plugin = Plugins().get("timevae")
>>> static, temporal, outcome = GoogleStocksDataloader(as_numpy=True).load()
```

(continues on next page)

(continued from previous page)

```
>>> loader = TimeSeriesDataLoader(
>>>             temporal_data=temporal_data,
>>>             static_data=static_data,
>>>             outcome=outcome,
>>> )
>>> plugin.fit(loader)
>>> plugin.generate()
```

class Config

Bases: object

arbitrary_types_allowed = True

validate_assignment = True

fit(*X: Union[*synthcity.plugins.core.dataloader.DataLoader*, pandas.core.frame.DataFrame]*, **args: Any*, ***kwargs: Any*) → Any

Training method the synthetic data plugin.

Parameters

- **X** – DataLoader. The reference dataset.
- **cond** – Optional, Union[pd.DataFrame, pd.Series, np.ndarray] Optional Training Conditional. The training conditional can be used to control to output of some models, like GANs or VAEs. The content can be anything, as long as it maps to the training dataset X. Usage example:

```
>>> from sklearn.datasets import load_iris
>>> from synthcity.plugins.core.dataloader import␣
↪GenericDataLoader
>>> from synthcity.plugins.core.constraints import Constraints
>>>
>>> # Load in `test_plugin` the generative model of choice
>>> # ....
>>>
>>> X, y = load_iris(as_frame=True, return_X_y=True)
>>> X["target"] = y
>>>
>>> X = GenericDataLoader(X)
>>> test_plugin.fit(X, cond=y)
>>>
>>> count = 10
>>> X_gen = test_plugin.generate(count, cond=np.ones(count))
>>>
>>> # The Conditional only optimizes the output generation
>>> # for GANs and VAEs, but does NOT guarantee the samples
>>> # are only from that condition.
>>> # If you want to guarantee that output contains only
>>> # "target" == 1 samples, use Constraints.
>>>
>>> constraints = Constraints(
>>>     rules=[
>>>         ("target", "==", 1),
>>>     ]
```

(continues on next page)

(continued from previous page)

```
>>> )
>>> X_gen = test_plugin.generate(count,
>>>         cond=np.ones(count),
>>>         constraints=constraints
>>>        )
>>> assert (X_gen["target"] == 1).all()
```

Returns self

classmethod fqdn() → str
The Fully-Qualified name of the plugin.

generate(*count: Optional[int] = None*, *constraints: Optional[*synthcity.plugins.core.constraints.Constraints*] = None*, *random_state: Optional[int] = None*, ***kwargs: Any*) → *synthcity.plugins.core.dataloader.DataLoader*
Synthetic data generation method.

Parameters

- **count** – optional int. The number of samples to generate. If None, it generated len(reference_dataset) samples.
- **cond** – Optional, Union[pd.DataFrame, pd.Series, np.ndarray]. Optional Generation Conditional. The conditional can be used only if the model was trained using a conditional too. If provided, it must have *count* length. Not all models support conditionals. The conditionals can be used in VAEs or GANs to speed-up the generation under some constraints. For model agnostic solutions, check out the *constraints* parameter.
- **constraints** – optional Constraints. Optional constraints to apply on the generated data. If none, the reference schema constraints are applied. The constraints are model agnostic, and will filter the output of the generative model. The constraints are a list of rules. Each rule is a tuple of the form (<feature>, <operation>, <value>).

  Valid Operations:

  - ”<”, “lt” : less than <value>
  - ”<=”, “le”: less or equal with <value>
  - ”>”, “gt” : greater than <value>
  - ”>=”, “ge”: greater or equal with <value>
  - ”==”, “eq”: equal with <value>
  - ”in”: valid for categorical features, and <value> must be array. for example, (“target”, “in”, [0, 1])
  - ”dtype”: <value> can be a data type. For example, (“target”, “dtype”, “int”)

  Usage example:

```
>>> from synthcity.plugins.core.constraints import Constraints
>>> constraints = Constraints(
>>>   rules=[
>>>             ("InterestingFeature", "==", 0),
>>>         ]
>>>     )
>>>
>>> syn_data = syn_model.generate(
```

(continues on next page)

(continued from previous page)

```
        count=count,
        constraints=constraints
    ).dataframe()
>>>
>>> assert (syn_data["InterestingFeature"] == 0).all()
```

- **random_state** – optional int. Optional random seed to use.

Returns <count> synthetic samples

static hyperparameter_space(***kwargs: Any*) → List[*synthcity.plugins.core.distribution.Distribution*]
Returns the hyperparameter space for the derived plugin.

static load(*buff: bytes*) → Any

static load_dict(*representation: dict*) → Any

static name() → str
The name of the plugin.

plot(*plt: Any*, *X:* synthcity.plugins.core.dataloader.DataLoader, *count: Optional[int] = None*, *plots: list = ['marginal', 'associations', 'tsne']*, ***kwargs: Any*) → Any
Plot the real-synthetic distributions.

Parameters

- **plt** – output
- **X** – DataLoader. The reference dataset.

Returns self

classmethod sample_hyperparameters(**args: Any*, ***kwargs: Any*) → Dict[str, Any]
Sample value from the hyperparameter space for the current plugin.

classmethod sample_hyperparameters_optuna(*trial: Any*, **args: Any*, ***kwargs: Any*) → Dict[str, Any]

save() → bytes

save_dict() → dict

save_to_file(*path: pathlib.Path*) → bytes

schema() → *synthcity.plugins.core.schema.Schema*
The reference schema

schema_includes(*other: Union[*synthcity.plugins.core.dataloader.DataLoader*, pandas.core.frame.DataFrame]*) → bool
Helper method to test if the reference schema includes a Dataset

Parameters **other** – DataLoader. The dataset to test

Returns bool, if the schema includes the dataset or not.

training_schema() → *synthcity.plugins.core.schema.Schema*
The internal schema

static type() → str
The type of the plugin.

static version() → str
API version

plugin
alias of *synthcity.plugins.time_series.plugin_timevae.TimeVAEPlugin*

10.1.6 Images

synthcity.plugins.images.plugin_image_cgan module

class ImageCGANPlugin(*n_units_latent: int = 100*, *n_iter: int = 1000*, *generator_nonlin: str = 'relu'*, *generator_dropout: float = 0.1*, *generator_n_residual_units: int = 2*, *discriminator_nonlin: str = 'leaky_relu'*, *discriminator_n_iter: int = 5*, *discriminator_dropout: float = 0.1*, *discriminator_n_residual_units: int = 2*, *lr: float = 0.0002*, *weight_decay: float = 0.001*, *opt_betas: tuple = (0.5, 0.999)*, *batch_size: int = 200*, *random_state: int = 0*, *clipping_value: int = 1*, *lambda_gradient_penalty: float = 10*, *device: Any = device(type='cpu')*, *patience: int = 5*, *patience_metric: Optional[synthcity.metrics.weighted_metrics.WeightedMetrics] = None*, *n_iter_print: int = 50*, *n_iter_min: int = 100*, *plot_progress: int = False*, *early_stopping: bool = True*, *workspace: pathlib.Path = PosixPath('workspace')*, *sampling_patience: int = 500*, ***kwargs: Any*)

Bases: *synthcity.plugins.core.plugin.Plugin*

Image (Conditional) GAN

Parameters

- **n_units_latent** – int The noise units size used by the generator.
- **n_iter** – int Maximum number of iterations in the Generator.
- **generator_nonlin** – string, default 'leaky_relu' Nonlinearity to use in the generator. Can be 'elu', 'relu', 'selu' or 'leaky_relu'.
- **generator_dropout** – float Dropout value. If 0, the dropout is not used.
- **generator_n_residual_units** – int The number of convolutions in residual units for the generator, 0 means no residual units
- **discriminator_nonlin** – string, default 'leaky_relu' Nonlinearity to use in the discriminator. Can be 'elu', 'relu', 'selu' or 'leaky_relu'.
- **discriminator_n_iter** – int Maximum number of iterations in the discriminator.
- **discriminator_dropout** – float Dropout value for the discriminator. If 0, the dropout is not used.
- **discriminator_n_residual_units** – int The number of convolutions in residual units for the discriminator, 0 means no residual units
- **parameters** (*# training*) –
- **lr** – float learning rate for optimizer..

- **weight_decay** – float l2 (ridge) penalty for the weights.
- **batch_size** – int Batch size
- **random_state** – int random seed to use
- **clipping_value** – int, default 0 Gradients clipping value. Zero disables the feature
- **device** – torch device Device: cpu or cuda
- **plot_progress** – bool Plot some synthetic samples every *n_iter_print*
- **stopping** (*# early*) –
- **n_iter_print** – int Number of iterations after which to print updates and check the validation loss.
- **n_iter_min** – int Minimum number of iterations to go through before starting early stopping
- **patience** – int Max number of iterations without any improvement before training early stopping is trigged.
- **patience_metric** – Optional[WeightedMetrics] If not None, the metric is used for evaluation the criterion for training early stopping.
- **early_stopping** – bool Evaluate the quality of the synthetic data using *patience_metric*, and stop after *patience* iteration with no improvement.
- **arguments** (*# Core Plugin*) –
- **workspace** – Path. Optional Path for caching intermediary results.

Example

```
>>> from torchvision import datasets
>>> from synthcity.plugins import Plugins
>>> from synthcity.plugins.core.dataloader import ImageDataLoader
>>>
>>> model = Plugins().get("image_cgan", n_iter = 10)
>>>
>>> dataset = datasets.MNIST(".", download=True)
>>> dataloader = ImageDataLoader(dataset).sample(100)
>>>
>>> model.fit(dataloader)
>>>
>>> X_gen = model.generate(50)
>>> assert len(X_gen) == 50
```

class Config

Bases: object

arbitrary_types_allowed = True

validate_assignment = True

fit(*X: Union[*synthcity.plugins.core.dataloader.DataLoader*, pandas.core.frame.DataFrame]*, **args: Any*, ***kwargs: Any*) → Any

Training method the synthetic data plugin.

Parameters

- **X** – DataLoader. The reference dataset.
- **cond** – Optional, Union[pd.DataFrame, pd.Series, np.ndarray] Optional Training Conditional. The training conditional can be used to control to output of some models, like GANs or VAEs. The content can be anything, as long as it maps to the training dataset X. Usage example:

```
>>> from sklearn.datasets import load_iris
>>> from synthcity.plugins.core.dataloader import␣
↪GenericDataLoader
>>> from synthcity.plugins.core.constraints import Constraints
>>>
>>> # Load in `test_plugin` the generative model of choice
>>> # ....
>>>
>>> X, y = load_iris(as_frame=True, return_X_y=True)
>>> X["target"] = y
>>>
>>> X = GenericDataLoader(X)
>>> test_plugin.fit(X, cond=y)
>>>
>>> count = 10
>>> X_gen = test_plugin.generate(count, cond=np.ones(count))
>>>
>>> # The Conditional only optimizes the output generation
>>> # for GANs and VAEs, but does NOT guarantee the samples
>>> # are only from that condition.
>>> # If you want to guarantee that output contains only
>>> # "target" == 1 samples, use Constraints.
>>>
>>> constraints = Constraints(
>>>     rules=[
>>>         ("target", "==", 1),
>>>     ]
>>> )
>>> X_gen = test_plugin.generate(count,
>>>         cond=np.ones(count),
>>>         constraints=constraints
>>>        )
>>> assert (X_gen["target"] == 1).all()
```

Returns self

classmethod fqdn() → str
The Fully-Qualified name of the plugin.

generate(*count: Optional[int] = None*, *constraints: Optional[*synthcity.plugins.core.constraints.Constraints*] = None*, *random_state: Optional[int] = None*, ***kwargs: Any*) → *synthcity.plugins.core.dataloader.DataLoader*
Synthetic data generation method.

Parameters

- **count** – optional int. The number of samples to generate. If None, it generated len(reference_dataset) samples.
- **cond** – Optional, Union[pd.DataFrame, pd.Series, np.ndarray]. Optional Generation

Conditional. The conditional can be used only if the model was trained using a conditional too. If provided, it must have *count* length. Not all models support conditionals. The conditionals can be used in VAEs or GANs to speed-up the generation under some constraints. For model agnostic solutions, check out the *constraints* parameter.

- **constraints** – optional Constraints. Optional constraints to apply on the generated data. If none, the reference schema constraints are applied. The constraints are model agnostic, and will filter the output of the generative model. The constraints are a list of rules. Each rule is a tuple of the form (<feature>, <operation>, <value>).

  Valid Operations:

  - "<", "lt" : less than <value>
  - "<=", "le": less or equal with <value>
  - ">", "gt" : greater than <value>
  - ">=", "ge": greater or equal with <value>
  - "==", "eq": equal with <value>
  - "in": valid for categorical features, and <value> must be array. for example, ("target", "in", [0, 1])
  - "dtype": <value> can be a data type. For example, ("target", "dtype", "int")

  Usage example:

```
>>> from synthcity.plugins.core.constraints import Constraints
>>> constraints = Constraints(
>>>   rules=[
>>>             ("InterestingFeature", "==", 0),
>>>         ]
>>>     )
>>>
>>> syn_data = syn_model.generate(
        count=count,
        constraints=constraints
    ).dataframe()
>>>
>>> assert (syn_data["InterestingFeature"] == 0).all()
```

- **random_state** – optional int. Optional random seed to use.

Returns <count> synthetic samples

static hyperparameter_space(***kwargs: Any*) → List[*synthcity.plugins.core.distribution.Distribution*]
Returns the hyperparameter space for the derived plugin.

static load(*buff: bytes*) → Any

static load_dict(*representation: dict*) → Any

static name() → str
The name of the plugin.

plot(*plt: Any*, *X:* synthcity.plugins.core.dataloader.DataLoader, *count: Optional[int] = None*, *plots: list = ['marginal', 'associations', 'tsne']*, ***kwargs: Any*) → Any
Plot the real-synthetic distributions.

Parameters

- **plt** – output
- **X** – DataLoader. The reference dataset.

Returns self

classmethod sample_hyperparameters(**args: Any*, ***kwargs: Any*) → Dict[str, Any]
Sample value from the hyperparameter space for the current plugin.

classmethod sample_hyperparameters_optuna(*trial: Any*, **args: Any*, ***kwargs: Any*) → Dict[str, Any]

save() → bytes

save_dict() → dict

save_to_file(*path: pathlib.Path*) → bytes

schema() → *synthcity.plugins.core.schema.Schema*
The reference schema

schema_includes(*other: Union[*synthcity.plugins.core.dataloader.DataLoader*, pandas.core.frame.DataFrame]*) → bool
Helper method to test if the reference schema includes a Dataset

Parameters **other** – DataLoader. The dataset to test

Returns bool, if the schema includes the dataset or not.

training_schema() → *synthcity.plugins.core.schema.Schema*
The internal schema

static type() → str
The type of the plugin.

static version() → str
API version

plugin
alias of *synthcity.plugins.images.plugin_image_cgan.ImageCGANPlugin*

synthcity.plugins.images.plugin_image_adsgan module

class ImageAdsGANPlugin(*n_units_latent: int = 100*, *n_iter: int = 1000*, *generator_nonlin: str = 'relu'*, *generator_dropout: float = 0.1*, *generator_n_residual_units: int = 2*, *discriminator_nonlin: str = 'leaky_relu'*, *discriminator_n_iter: int = 5*, *discriminator_dropout: float = 0.1*, *discriminator_n_residual_units: int = 2*, *lr: float = 0.0002*, *weight_decay: float = 0.001*, *opt_betas: tuple = (0.5, 0.999)*, *batch_size: int = 200*, *random_state: int = 0*, *clipping_value: int = 1*, *lambda_gradient_penalty: float = 10*, *lambda_identifiability_penalty: float = 0.1*, *device: Any = device(type='cpu')*, *patience: int = 5*, *patience_metric: Optional[*synthcity.metrics.weighted_metrics.WeightedMetrics*] = None*, *n_iter_print: int = 50*, *n_iter_min: int = 100*, *plot_progress: int = False*, *early_stopping: bool = True*, *workspace: pathlib.Path = PosixPath('workspace')*, *sampling_patience: int = 500*, ***kwargs: Any*)
Bases: *synthcity.plugins.core.plugin.Plugin*

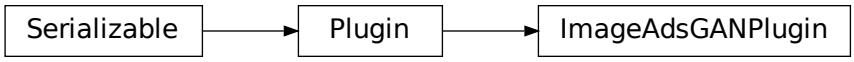


Image AdsGAN - Anonymization through Data Synthesis using Generative Adversarial Networks.

Parameters

- **n_units_latent** – int The noise units size used by the generator.
- **n_iter** – int Maximum number of iterations in the Generator.
- **generator_nonlin** – string, default 'leaky_relu' Nonlinearity to use in the generator. Can be 'elu', 'relu', 'selu' or 'leaky_relu'.
- **generator_dropout** – float Dropout value. If 0, the dropout is not used.
- **generator_n_residual_units** – int The number of convolutions in residual units for the generator, 0 means no residual units
- **discriminator_nonlin** – string, default 'leaky_relu' Nonlinearity to use in the discriminator. Can be 'elu', 'relu', 'selu' or 'leaky_relu'.
- **discriminator_n_iter** – int Maximum number of iterations in the discriminator.
- **discriminator_dropout** – float Dropout value for the discriminator. If 0, the dropout is not used.
- **discriminator_n_residual_units** – int The number of convolutions in residual units for the discriminator, 0 means no residual units
- **parameters** (*# training*) –
- **lr** – float learning rate for optimizer
- **weight_decay** – float l2 (ridge) penalty for the weights.
- **batch_size** – int Batch size
- **random_state** – int random seed to use
- **clipping_value** – int, default 0 Gradients clipping value. Zero disables the feature
- **lambda_gradient_penalty** – float = 10 Weight for the gradient penalty
- **lambda_identifiability_penalty** – float = 0.1 Weight for the identifiability penalty
- **device** – torch device Device: cpu or cuda
- **plot_progress** – bool Plot some synthetic samples every *n_iter_print*
- **stopping** (*# early*) –
- **n_iter_print** – int Number of iterations after which to print updates and check the validation loss.
- **n_iter_min** – int Minimum number of iterations to go through before starting early stopping
- **early_stopping** – bool Evaluate the quality of the synthetic data using *patience_metric*, and stop after *patience* iteration with no improvement.

- **patience** – int Max number of iterations without any improvement before training early stopping is trigged.
- **patience_metric** – Optional[WeightedMetrics] If not None, the metric is used for evaluation the criterion for training early stopping.
- **arguments** (*# Core Plugin*) –
- **workspace** – Path. Optional Path for caching intermediary results.

Example

```
>>> from torchvision import datasets
>>> from synthcity.plugins import Plugins
>>> from synthcity.plugins.core.dataloader import ImageDataLoader
>>>
>>> model = Plugins().get("image_adsgan", n_iter = 10)
>>>
>>> dataset = datasets.MNIST(".", download=True)
>>> dataloader = ImageDataLoader(dataset).sample(100)
>>>
>>> model.fit(dataloader)
>>>
>>> X_gen = model.generate(50)
>>> assert len(X_gen) == 50
```

class Config

Bases: object

arbitrary_types_allowed = True

validate_assignment = True

fit(*X: Union[*synthcity.plugins.core.dataloader.DataLoader*, pandas.core.frame.DataFrame]*, **args: Any*, ***kwargs: Any*) → Any

Training method the synthetic data plugin.

Parameters

- **X** – DataLoader. The reference dataset.
- **cond** – Optional, Union[pd.DataFrame, pd.Series, np.ndarray] Optional Training Conditional. The training conditional can be used to control to output of some models, like GANs or VAEs. The content can be anything, as long as it maps to the training dataset X. Usage example:

```
>>> from sklearn.datasets import load_iris
>>> from synthcity.plugins.core.dataloader import␣
↪GenericDataLoader
>>> from synthcity.plugins.core.constraints import Constraints
>>>
>>> # Load in `test_plugin` the generative model of choice
>>> # ....
>>>
>>> X, y = load_iris(as_frame=True, return_X_y=True)
>>> X["target"] = y
>>>
```

(continues on next page)

(continued from previous page)

```
>>> X = GenericDataLoader(X)
>>> test_plugin.fit(X, cond=y)
>>>
>>> count = 10
>>> X_gen = test_plugin.generate(count, cond=np.ones(count))
>>>
>>> # The Conditional only optimizes the output generation
>>> # for GANs and VAEs, but does NOT guarantee the samples
>>> # are only from that condition.
>>> # If you want to guarantee that output contains only
>>> # "target" == 1 samples, use Constraints.
>>>
>>> constraints = Constraints(
>>>     rules=[
>>>         ("target", "==", 1),
>>>     ]
>>> )
>>> X_gen = test_plugin.generate(count,
>>>         cond=np.ones(count),
>>>         constraints=constraints
>>>        )
>>> assert (X_gen["target"] == 1).all()
```

Returns self

classmethod fqdn() → str
The Fully-Qualified name of the plugin.

generate(*count: Optional[int] = None*, *constraints: Optional[*synthcity.plugins.core.constraints.Constraints*] = None*, *random_state: Optional[int] = None*, ***kwargs: Any*) → *synthcity.plugins.core.dataloader.DataLoader*
Synthetic data generation method.

Parameters

- **count** – optional int. The number of samples to generate. If None, it generated len(reference_dataset) samples.
- **cond** – Optional, Union[pd.DataFrame, pd.Series, np.ndarray]. Optional Generation Conditional. The conditional can be used only if the model was trained using a conditional too. If provided, it must have *count* length. Not all models support conditionals. The conditionals can be used in VAEs or GANs to speed-up the generation under some constraints. For model agnostic solutions, check out the *constraints* parameter.
- **constraints** – optional Constraints. Optional constraints to apply on the generated data. If none, the reference schema constraints are applied. The constraints are model agnostic, and will filter the output of the generative model. The constraints are a list of rules. Each rule is a tuple of the form (<feature>, <operation>, <value>).

  Valid Operations:

  - "<", "lt" : less than <value>
  - "<=", "le": less or equal with <value>
  - ">", "gt" : greater than <value>
  - ">=", "ge": greater or equal with <value>

- "==", "eq": equal with <value>
- "in": valid for categorical features, and <value> must be array. for example, ("target", "in", [0, 1])
- "dtype": <value> can be a data type. For example, ("target", "dtype", "int")

Usage example:

```
>>> from synthcity.plugins.core.constraints import Constraints
>>> constraints = Constraints(
>>>   rules=[
>>>             ("InterestingFeature", "==", 0),
>>>         ]
>>>     )
>>>
>>> syn_data = syn_model.generate(
        count=count,
        constraints=constraints
    ).dataframe()
>>>
>>> assert (syn_data["InterestingFeature"] == 0).all()
```

- **random_state** – optional int. Optional random seed to use.

Returns <count> synthetic samples

static hyperparameter_space(***kwargs: Any*) → List[*synthcity.plugins.core.distribution.Distribution*]
Returns the hyperparameter space for the derived plugin.

static load(*buff: bytes*) → Any

static load_dict(*representation: dict*) → Any

static name() → str
The name of the plugin.

plot(*plt: Any*, *X:* synthcity.plugins.core.dataloader.DataLoader, *count: Optional[int] = None*, *plots: list = ['marginal', 'associations', 'tsne']*, ***kwargs: Any*) → Any
Plot the real-synthetic distributions.

Parameters

- **plt** – output
- **X** – DataLoader. The reference dataset.

Returns self

classmethod sample_hyperparameters(**args: Any*, ***kwargs: Any*) → Dict[str, Any]
Sample value from the hyperparameter space for the current plugin.

classmethod sample_hyperparameters_optuna(*trial: Any*, **args: Any*, ***kwargs: Any*) → Dict[str, Any]

save() → bytes

save_dict() → dict

save_to_file(*path: pathlib.Path*) → bytes

schema() → *synthcity.plugins.core.schema.Schema*
The reference schema

schema_includes(*other: Union[*synthcity.plugins.core.dataloader.DataLoader, *pandas.core.frame.DataFrame]*) → bool
Helper method to test if the reference schema includes a Dataset

Parameters **other** – DataLoader. The dataset to test

Returns bool, if the schema includes the dataset or not.

training_schema() → *synthcity.plugins.core.schema.Schema*
The internal schema

static type() → str
The type of the plugin.

static version() → str
API version

plugin
alias of *synthcity.plugins.images.plugin_image_adsgan.ImageAdsGANPlugin*

CHAPTER

ELEVEN

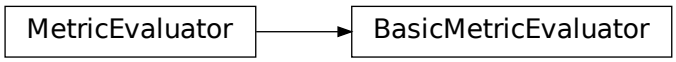
METRICS

11.1 Metrics and benchmarks

11.1.1 synthcity.metrics.eval_sanity module

class BasicMetricEvaluator(***kwargs: Any*)

Bases: synthcity.metrics.core.metric.MetricEvaluator



static direction() → str

evaluate(*X_gt:* synthcity.plugins.core.dataloader.DataLoader, *X_syn:* synthcity.plugins.core.dataloader.DataLoader) → float

evaluate_default(*X_gt:* synthcity.plugins.core.dataloader.DataLoader, *X_syn:* synthcity.plugins.core.dataloader.DataLoader) → float

classmethod fqdn() → str

static name() → str

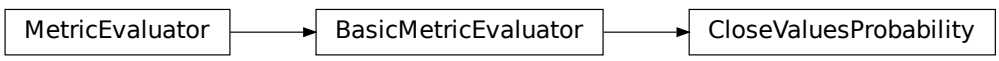
reduction() → Callable

static type() → str

use_cache(*path: pathlib.Path*) → bool

class CloseValuesProbability(***kwargs: Any*)

Bases: *synthcity.metrics.eval_sanity.BasicMetricEvaluator*



Compute the probability of close values between the real and synthetic data.

Score: 0 means there is no chance to have synthetic rows similar to the real. 1 means that all the synthetic rows are similar to some real rows.

static direction() → str

evaluate(*X_gt:* synthcity.plugins.core.dataloader.DataLoader, *X_syn:* synthcity.plugins.core.dataloader.DataLoader) → Dict

evaluate_default(*X_gt:* synthcity.plugins.core.dataloader.DataLoader, *X_syn:* synthcity.plugins.core.dataloader.DataLoader) → float

classmethod fqdn() → str

static name() → str

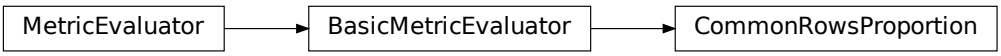
reduction() → Callable

static type() → str

use_cache(*path: pathlib.Path*) → bool

class CommonRowsProportion(***kwargs: Any*)

Bases: *synthcity.metrics.eval_sanity.BasicMetricEvaluator*



Returns the proportion of rows in the real dataset leaked in the synthetic dataset.

Score: 0: there are no common rows between the real and synthetic datasets. 1: all the rows in the real dataset are leaked in the synthetic dataset.

static direction() → str

evaluate(*X_gt:* synthcity.plugins.core.dataloader.DataLoader, *X_syn:* synthcity.plugins.core.dataloader.DataLoader) → Dict

evaluate_default(*X_gt:* synthcity.plugins.core.dataloader.DataLoader, *X_syn:* synthcity.plugins.core.dataloader.DataLoader) → float

classmethod fqdn() → str

static name() → str

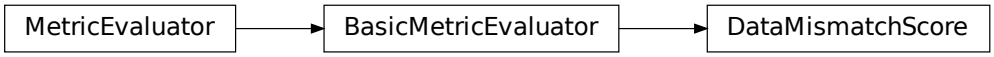
reduction() → Callable

static type() → str

use_cache(*path: pathlib.Path*) → bool

class DataMismatchScore(***kwargs: Any*)

Bases: *synthcity.metrics.eval_sanity.BasicMetricEvaluator*



Basic sanity score. Compares the data types between the column of the ground truth and the synthetic data.

Score: 0: no datatype mismatch. 1: complete data type mismatch between the datasets.

static direction() → str

evaluate(*X_gt:* synthcity.plugins.core.dataloader.DataLoader, *X_syn:* synthcity.plugins.core.dataloader.DataLoader) → Dict

evaluate_default(*X_gt:* synthcity.plugins.core.dataloader.DataLoader, *X_syn:* synthcity.plugins.core.dataloader.DataLoader) → float

classmethod fqdn() → str

static name() → str

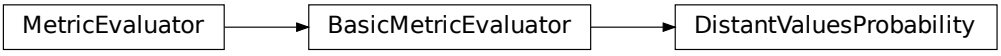
reduction() → Callable

static type() → str

use_cache(*path: pathlib.Path*) → bool

class DistantValuesProbability(***kwargs: Any*)

Bases: *synthcity.metrics.eval_sanity.BasicMetricEvaluator*



Compute the probability of distant values between the real and synthetic data.

Score: 0 means there is no chance to have rows in the synthetic far away from the real data. 1 means all the synthetic datapoints are far away from the real data.

static direction() → str

evaluate(*X_gt:* synthcity.plugins.core.dataloader.DataLoader, *X_syn:* synthcity.plugins.core.dataloader.DataLoader) → Dict

evaluate_default(*X_gt:* synthcity.plugins.core.dataloader.DataLoader, *X_syn:* synthcity.plugins.core.dataloader.DataLoader) → float

classmethod fqdn() → str

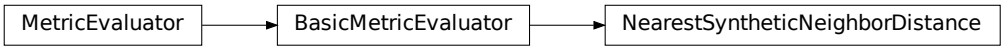
static name() → str

reduction() → Callable

static type() → str

use_cache(*path: pathlib.Path*) → bool

class NearestSyntheticNeighborDistance(***kwargs: Any*)
Bases: *synthcity.metrics.eval_sanity.BasicMetricEvaluator*



Computes the <reduction>(distance) from the real data to the closest neighbor in the synthetic data

static direction() → str

evaluate(*X_gt:* synthcity.plugins.core.dataloader.DataLoader, *X_syn:* synthcity.plugins.core.dataloader.DataLoader) → Dict

evaluate_default(*X_gt:* synthcity.plugins.core.dataloader.DataLoader, *X_syn:* synthcity.plugins.core.dataloader.DataLoader) → float

classmethod fqdn() → str

static name() → str

reduction() → Callable

static type() → str

use_cache(*path: pathlib.Path*) → bool

11.1.2 synthcity.metrics.eval_statistical module

class AlphaPrecision(***kwargs: Any*)
Bases: *synthcity.metrics.eval_statistical.StatisticalEvaluator*

Evaluates the alpha-precision, beta-recall, and authenticity scores.

The class evaluates the synthetic data using a tuple of three metrics: alpha-precision, beta-recall, and authenticity. Note that these metrics can be evaluated for each synthetic data point (which are useful for auditing and post-processing). Here we average the scores to reflect the overall quality of the data. The formal definitions can be found in the reference below:

Alaa, Ahmed, Boris Van Breugel, Evgeny S. Saveliev, and Mihaela van der Schaar. "How faithful is your synthetic data? sample-level metrics for evaluating and auditing generative models." In International Conference on Machine Learning, pp. 290-306. PMLR, 2022.

static direction() → str

evaluate(*X_gt:* synthcity.plugins.core.dataloader.DataLoader, *X_syn:* synthcity.plugins.core.dataloader.DataLoader) → Dict

evaluate_default(*X_gt:* synthcity.plugins.core.dataloader.DataLoader, *X_syn:* synthcity.plugins.core.dataloader.DataLoader) → float

classmethod fqdn() → str

metrics(*X: numpy.ndarray*, *X_syn: numpy.ndarray*, *emb_center: Optional[numpy.ndarray] = None*) → Tuple

static name() → str

reduction() → Callable

static type() → str

use_cache(*path: pathlib.Path*) → bool

class ChiSquaredTest(***kwargs: Any*)

Bases: *synthcity.metrics.eval_statistical.StatisticalEvaluator*

Performs the one-way chi-square test.

Returns The p-value. A small value indicates that we can reject the null hypothesis and that the distributions are different.

Score: 0: the distributions are different 1: the distributions are identical.

static direction() → str

evaluate(*X_gt:* synthcity.plugins.core.dataloader.DataLoader, *X_syn:* synthcity.plugins.core.dataloader.DataLoader) → Dict

evaluate_default(*X_gt:* synthcity.plugins.core.dataloader.DataLoader, *X_syn:* synthcity.plugins.core.dataloader.DataLoader) → float

classmethod fqdn() → str

static name() → str

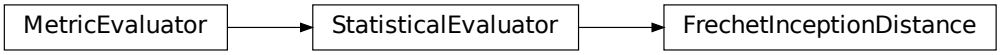
reduction() → Callable

static type() → str

use_cache(*path: pathlib.Path*) → bool

class FrechetInceptionDistance(***kwargs: Any*)

Bases: *synthcity.metrics.eval_statistical.StatisticalEvaluator*



Calculates the Frechet Inception Distance (FID) to evalulate GANs.

Paper: GANs Trained by a Two Time-Scale Update Rule Converge to a Local Nash Equilibrium.

The FID metric calculates the distance between two distributions of images. Typically, we have summary statistics (mean & covariance matrix) of one of these distributions, while the 2nd distribution is given by a GAN.

Adapted by Boris van Breugel(bv292@cam.ac.uk)

static direction() → str

evaluate(*X_gt:* synthcity.plugins.core.dataloader.DataLoader, *X_syn:* synthcity.plugins.core.dataloader.DataLoader) → Dict

evaluate_default(*X_gt:* synthcity.plugins.core.dataloader.DataLoader, *X_syn:* synthcity.plugins.core.dataloader.DataLoader) → float

classmethod fqdn() → str

static name() → str

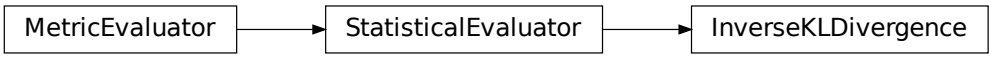
reduction() → Callable

static type() → str

use_cache(*path: pathlib.Path*) → bool

class InverseKLDivergence(***kwargs: Any*)

Bases: *synthcity.metrics.eval_statistical.StatisticalEvaluator*



Returns the average inverse of the Kullback–Leibler Divergence metric.

Score: 0: the datasets are from different distributions. 1: the datasets are from the same distribution.

static direction() → str

evaluate(*X_gt:* synthcity.plugins.core.dataloader.DataLoader, *X_syn:* synthcity.plugins.core.dataloader.DataLoader) → Dict

evaluate_default(*X_gt:* synthcity.plugins.core.dataloader.DataLoader, *X_syn:* synthcity.plugins.core.dataloader.DataLoader) → float

classmethod fqdn() → str

static name() → str

reduction() → Callable

static type() → str

use_cache(*path: pathlib.Path*) → bool

class JensenShannonDistance(*normalize: bool = True*, ***kwargs: Any*)
Bases: *synthcity.metrics.eval_statistical.StatisticalEvaluator*

Evaluate the average Jensen-Shannon distance (metric) between two probability arrays.

static direction() → str

evaluate(*X_gt:* synthcity.plugins.core.dataloader.DataLoader, *X_syn:* synthcity.plugins.core.dataloader.DataLoader) → Dict

evaluate_default(*X_gt:* synthcity.plugins.core.dataloader.DataLoader, *X_syn:* synthcity.plugins.core.dataloader.DataLoader) → float

classmethod fqdn() → str

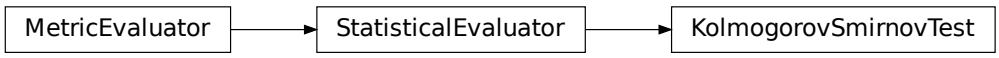
static name() → str

reduction() → Callable

static type() → str

use_cache(*path: pathlib.Path*) → bool

class KolmogorovSmirnovTest(***kwargs: Any*)
Bases: *synthcity.metrics.eval_statistical.StatisticalEvaluator*



Performs the Kolmogorov-Smirnov test for goodness of fit.

Score: 0: the distributions are totally different. 1: the distributions are identical.

static direction() → str

evaluate(*X_gt:* synthcity.plugins.core.dataloader.DataLoader, *X_syn:* synthcity.plugins.core.dataloader.DataLoader) → Dict

evaluate_default(*X_gt:* synthcity.plugins.core.dataloader.DataLoader, *X_syn:* synthcity.plugins.core.dataloader.DataLoader) → float

classmethod fqdn() → str

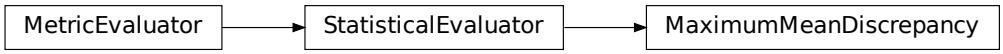
static name() → str

reduction() → Callable

static type() → str

use_cache(*path: pathlib.Path*) → bool

class MaximumMeanDiscrepancy(*kernel: str = 'rbf'*, ***kwargs: Any*)
Bases: *synthcity.metrics.eval_statistical.StatisticalEvaluator*



Empirical maximum mean discrepancy. The lower the result the more evidence that distributions are the same.

Parameters **kernel** – "rbf", "linear" or "polynomial"

Score: 0: The distributions are the same. 1: The distributions are totally different.

static direction() → str

evaluate(*X_gt:* synthcity.plugins.core.dataloader.DataLoader, *X_syn:* synthcity.plugins.core.dataloader.DataLoader) → Dict

evaluate_default(*X_gt:* synthcity.plugins.core.dataloader.DataLoader, *X_syn:* synthcity.plugins.core.dataloader.DataLoader) → float

classmethod fqdn() → str

static name() → str

reduction() → Callable

static type() → str

use_cache(*path: pathlib.Path*) → bool

class PRDCScore(*nearest_k: int = 5*, ***kwargs: Any*)

Bases: *synthcity.metrics.eval_statistical.StatisticalEvaluator*

Computes precision, recall, density, and coverage given two manifolds.

Parameters **nearest_k** – int.

static direction() → str

evaluate(*X_gt:* synthcity.plugins.core.dataloader.DataLoader, *X_syn:* synthcity.plugins.core.dataloader.DataLoader) → Dict

evaluate_default(*X_gt:* synthcity.plugins.core.dataloader.DataLoader, *X_syn:* synthcity.plugins.core.dataloader.DataLoader) → float

classmethod fqdn() → str

static name() → str

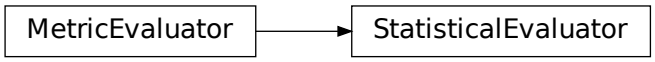
reduction() → Callable

static type() → str

use_cache(*path: pathlib.Path*) → bool

class StatisticalEvaluator(***kwargs: Any*)

Bases: synthcity.metrics.core.metric.MetricEvaluator



abstract static direction() → str

evaluate(*X_gt:* synthcity.plugins.core.dataloader.DataLoader, *X_syn:* synthcity.plugins.core.dataloader.DataLoader) → Dict

evaluate_default(*X_gt:* synthcity.plugins.core.dataloader.DataLoader, *X_syn:* synthcity.plugins.core.dataloader.DataLoader) → float

classmethod fqdn() → str

abstract static name() → str

reduction() → Callable

static type() → str

use_cache(*path: pathlib.Path*) → bool

class SurvivalKMDistance(***kwargs: Any*)

Bases: *synthcity.metrics.eval_statistical.StatisticalEvaluator*

The distance between two Kaplan-Meier plots. Used for survival analysis

static direction() → str

evaluate(*X_gt:* synthcity.plugins.core.dataloader.DataLoader, *X_syn:* synthcity.plugins.core.dataloader.DataLoader) → Dict

evaluate_default(*X_gt:* synthcity.plugins.core.dataloader.DataLoader, *X_syn:* synthcity.plugins.core.dataloader.DataLoader) → float

classmethod fqdn() → str

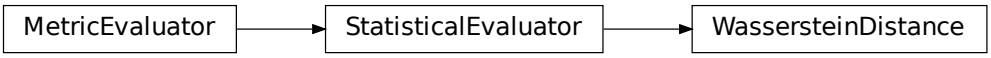
static name() → str

reduction() → Callable

static type() → str

use_cache(*path: pathlib.Path*) → bool

class WassersteinDistance(***kwargs: Any*)
Bases: *synthcity.metrics.eval_statistical.StatisticalEvaluator*



Compare Wasserstein distance between original data and synthetic data.

Parameters

- **X** – original data
- **X_syn** – synthetically generated data

Returns Wasserstein distance

Return type WD_value

static direction() → str

evaluate(*X_gt:* synthcity.plugins.core.dataloader.DataLoader, *X_syn:* synthcity.plugins.core.dataloader.DataLoader) → Dict

evaluate_default(*X_gt:* synthcity.plugins.core.dataloader.DataLoader, *X_syn:* synthcity.plugins.core.dataloader.DataLoader) → float

classmethod fqdn() → str

static name() → str

reduction() → Callable

static type() → str

use_cache(*path: pathlib.Path*) → bool

11.1.3 synthcity.metrics.eval_performance module

class AugmentationPerformanceEvaluatorLinear(***kwargs: Any*)
Bases: *synthcity.metrics.eval_performance.PerformanceEvaluatorLinear*

static direction() → str

evaluate(*X_gt:* synthcity.plugins.core.dataloader.DataLoader, *X_syn:* synthcity.plugins.core.dataloader.DataLoader) → Dict

evaluate_default(*X_gt:* synthcity.plugins.core.dataloader.DataLoader, *X_syn:* synthcity.plugins.core.dataloader.DataLoader) → float

classmethod fqdn() → str

static name() → str

reduction() → Callable

static standard_performance_output_keys() → List

static type() → str

use_cache(*path: pathlib.Path*) → bool

class AugmentationPerformanceEvaluatorMLP(***kwargs: Any*)

Bases: *synthcity.metrics.eval_performance.PerformanceEvaluatorMLP*

static direction() → str

evaluate(*X_gt:* synthcity.plugins.core.dataloader.DataLoader, *X_syn:* synthcity.plugins.core.dataloader.DataLoader) → Dict

evaluate_default(*X_gt:* synthcity.plugins.core.dataloader.DataLoader, *X_syn:* synthcity.plugins.core.dataloader.DataLoader) → float

classmethod fqdn() → str

static name() → str

reduction() → Callable

static standard_performance_output_keys() → List

static type() → str

use_cache(*path: pathlib.Path*) → bool

class AugmentationPerformanceEvaluatorXGB(***kwargs: Any*)

Bases: *synthcity.metrics.eval_performance.PerformanceEvaluatorXGB*

static direction() → str

evaluate(*X_gt:* synthcity.plugins.core.dataloader.DataLoader, *X_syn:* synthcity.plugins.core.dataloader.DataLoader) → Dict

evaluate_default(*X_gt:* synthcity.plugins.core.dataloader.DataLoader, *X_syn:* synthcity.plugins.core.dataloader.DataLoader) → float

classmethod fqdn() → str

static name() → str

reduction() → Callable

static standard_performance_output_keys() → List

static type() → str

use_cache(*path: pathlib.Path*) → bool

class FeatureImportanceRankDistance(*distance: str = 'kendall', **kwargs: Any*)

Bases: synthcity.metrics.core.metric.MetricEvaluator

Train an XGBoost classifier or regressor on the synthetic data and evaluate the feature importance. Train an XGBoost model on the real data and evaluate the feature importance.

Returns the rank distance between the feature importance Returns the average performance discrepancy between training on real data vs on synthetic data.

Score: close to 1: similar performance close to 0: unrelated close to -1: the ranks have different monotony.

static direction() → str

distance(*lhs: numpy.ndarray*, *rhs: numpy.ndarray*) → Tuple[float, float]

evaluate(*X_gt:* synthcity.plugins.core.dataloader.DataLoader, *X_syn:* synthcity.plugins.core.dataloader.DataLoader) → Dict

evaluate_default(*X_gt:* synthcity.plugins.core.dataloader.DataLoader, *X_syn:* synthcity.plugins.core.dataloader.DataLoader) → float

classmethod fqdn() → str

static name() → str

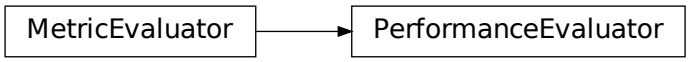
reduction() → Callable

static type() → str

use_cache(*path: pathlib.Path*) → bool

class PerformanceEvaluator(***kwargs: Any*)

Bases: synthcity.metrics.core.metric.MetricEvaluator



Evaluating synthetic data based on downstream performance.

This implements the train-on-synthetic test-on-real methodology for evaluation.

static direction() → str

abstract evaluate(*X_gt:* synthcity.plugins.core.dataloader.DataLoader, *X_syn:* synthcity.plugins.core.dataloader.DataLoader) → Dict

evaluate_default(*X_gt:* synthcity.plugins.core.dataloader.DataLoader, *X_syn:* synthcity.plugins.core.dataloader.DataLoader) → float

classmethod fqdn() → str

abstract static name() → str

reduction() → Callable

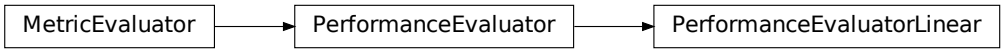
static standard_performance_output_keys() → List

static type() → str

use_cache(*path: pathlib.Path*) → bool

class PerformanceEvaluatorLinear(***kwargs: Any*)

Bases: *synthcity.metrics.eval_performance.PerformanceEvaluator*



Train a Linear classifier or regressor on the synthetic data and evaluate the performance on real test data.

Returns the average performance discrepancy between training on real data vs on synthetic data.

Score: close to 1: similar performance close to 0: massive performance degradation

static direction() → str

evaluate(*X_gt:* synthcity.plugins.core.dataloader.DataLoader, *X_syn:* synthcity.plugins.core.dataloader.DataLoader) → Dict

evaluate_default(*X_gt:* synthcity.plugins.core.dataloader.DataLoader, *X_syn:* synthcity.plugins.core.dataloader.DataLoader) → float

classmethod fqdn() → str

static name() → str

reduction() → Callable

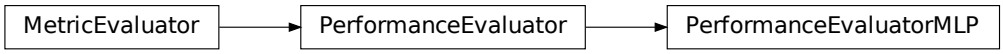
static standard_performance_output_keys() → List

static type() → str

use_cache(*path: pathlib.Path*) → bool

class PerformanceEvaluatorMLP(***kwargs: Any*)

Bases: *synthcity.metrics.eval_performance.PerformanceEvaluator*



Train a Neural Net classifier or regressor on the synthetic data and evaluate the performance on real test data.

Returns the average performance discrepancy between training on real data vs on synthetic data.

Score: close to 1: similar performance close to 1: massive performance degradation

static direction() → str

evaluate(*X_gt:* synthcity.plugins.core.dataloader.DataLoader, *X_syn:* synthcity.plugins.core.dataloader.DataLoader) → Dict

evaluate_default(*X_gt:* synthcity.plugins.core.dataloader.DataLoader, *X_syn:* synthcity.plugins.core.dataloader.DataLoader) → float

classmethod fqdn() → str

static name() → str

reduction() → Callable

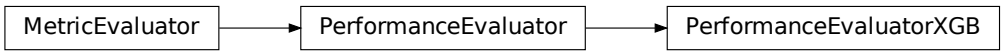
static standard_performance_output_keys() → List

static type() → str

use_cache(*path: pathlib.Path*) → bool

class PerformanceEvaluatorXGB(***kwargs: Any*)

Bases: *synthcity.metrics.eval_performance.PerformanceEvaluator*



Train an XGBoost classifier or regressor on the synthetic data and evaluate the performance on real test data.

Returns the average performance discrepancy between training on real data vs on synthetic data.

Score: close to 1: similar performance close to 0: massive performance degradation

static direction() → str

evaluate(*X_gt:* synthcity.plugins.core.dataloader.DataLoader, *X_syn:* synthcity.plugins.core.dataloader.DataLoader) → Dict

evaluate_default(*X_gt:* synthcity.plugins.core.dataloader.DataLoader, *X_syn:* synthcity.plugins.core.dataloader.DataLoader) → float

classmethod fqdn() → str

static name() → str

reduction() → Callable

static standard_performance_output_keys() → List

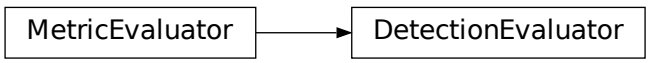
static type() → str

use_cache(*path: pathlib.Path*) → bool

11.1.4 synthcity.metrics.eval_detection module

class DetectionEvaluator(***kwargs: Any*)

Bases: synthcity.metrics.core.metric.MetricEvaluator



Train a SKLearn classifier to detect the synthetic data from real data.

Synthetic and real data are combined to form a new dataset. K-fold cross validation is performed to see how well a classifier can distinguish real from synthetic.

Returns The average AUCROC score for detecting synthetic data.

Score: 0: The datasets are indistinguishable. 1: The datasets are totally distinguishable.

static direction() → str

evaluate(*X_gt:* synthcity.plugins.core.dataloader.DataLoader, *X_syn:* synthcity.plugins.core.dataloader.DataLoader) → Dict

evaluate_default(*X_gt:* synthcity.plugins.core.dataloader.DataLoader, *X_syn:* synthcity.plugins.core.dataloader.DataLoader) → float

classmethod fqdn() → str

static name() → str

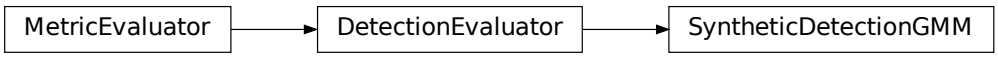
reduction() → Callable

static type() → str

use_cache(*path: pathlib.Path*) → bool

class SyntheticDetectionGMM(***kwargs: Any*)

Bases: *synthcity.metrics.eval_detection.DetectionEvaluator*



Train a GaussianMixture model to detect synthetic data.

Returns The average score for detecting synthetic data.

Score: 0: The datasets are indistinguishable. 1: The datasets are totally distinguishable.

static direction() → str

evaluate(*X_gt:* synthcity.plugins.core.dataloader.DataLoader, *X_syn:* synthcity.plugins.core.dataloader.DataLoader) → Dict

evaluate_default(*X_gt:* synthcity.plugins.core.dataloader.DataLoader, *X_syn:* synthcity.plugins.core.dataloader.DataLoader) → float

classmethod fqdn() → str

static name() → str

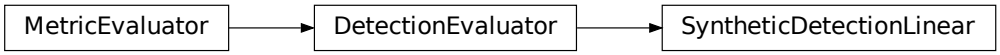
reduction() → Callable

static type() → str

use_cache(*path: pathlib.Path*) → bool

class SyntheticDetectionLinear(***kwargs: Any*)

Bases: *synthcity.metrics.eval_detection.DetectionEvaluator*



Train a LogisticRegression classifier to detect the synthetic data.

Returns The average AUCROC score for detecting synthetic data.

Score: 0: The datasets are indistinguishable. 1: The datasets are totally distinguishable.

static direction() → str

evaluate(*X_gt:* synthcity.plugins.core.dataloader.DataLoader, *X_syn:* synthcity.plugins.core.dataloader.DataLoader) → Dict

evaluate_default(*X_gt:* synthcity.plugins.core.dataloader.DataLoader, *X_syn:* synthcity.plugins.core.dataloader.DataLoader) → float

classmethod fqdn() → str

static name() → str

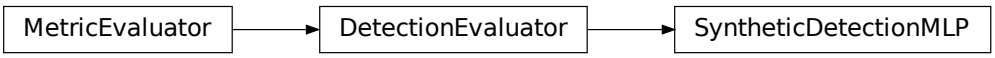
reduction() → Callable

static type() → str

use_cache(*path: pathlib.Path*) → bool

class SyntheticDetectionMLP(***kwargs: Any*)

Bases: *synthcity.metrics.eval_detection.DetectionEvaluator*



Train a MLP classifier to detect the synthetic data.

Returns The average AUCROC score for detecting synthetic data.

Score: 0: The datasets are indistinguishable. 1: The datasets are totally distinguishable.

static direction() → str

evaluate(*X_gt:* synthcity.plugins.core.dataloader.DataLoader, *X_syn:* synthcity.plugins.core.dataloader.DataLoader) → Dict

evaluate_default(*X_gt:* synthcity.plugins.core.dataloader.DataLoader, *X_syn:* synthcity.plugins.core.dataloader.DataLoader) → float

classmethod fqdn() → str

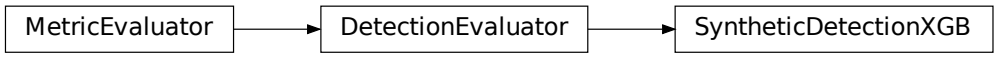
static name() → str

reduction() → Callable

static type() → str

use_cache(*path: pathlib.Path*) → bool

class SyntheticDetectionXGB(***kwargs: Any*)
Bases: *synthcity.metrics.eval_detection.DetectionEvaluator*



Train a XGBoostclassifier to detect the synthetic data.

Returns The average AUCROC score for detecting synthetic data.

Score: 0: The datasets are indistinguishable. 1: The datasets are totally distinguishable.

static direction() → str

evaluate(*X_gt:* synthcity.plugins.core.dataloader.DataLoader, *X_syn:* synthcity.plugins.core.dataloader.DataLoader) → Dict

evaluate_default(*X_gt:* synthcity.plugins.core.dataloader.DataLoader, *X_syn:* synthcity.plugins.core.dataloader.DataLoader) → float

classmethod fqdn() → str

static name() → str

reduction() → Callable

static type() → str

use_cache(*path: pathlib.Path*) → bool

11.1.5 synthcity.metrics.eval_privacy module

class DeltaPresence(***kwargs: Any*)
Bases: *synthcity.metrics.eval_privacy.PrivacyEvaluator*

Returns the maximum re-identification probability on the real dataset from the synthetic dataset.

For each dataset partition, we report the maximum ratio of unique sensitive information between the real dataset and in the synthetic dataset.

static direction() → str

evaluate(*X_gt:* synthcity.plugins.core.dataloader.DataLoader, *X_syn:* synthcity.plugins.core.dataloader.DataLoader, **args: Any*, ***kwargs: Any*) → Dict

evaluate_default(*X_gt:* synthcity.plugins.core.dataloader.DataLoader, *X_syn:* synthcity.plugins.core.dataloader.DataLoader) → float

classmethod fqdn() → str

static name() → str

reduction() → Callable

static type() → str

use_cache(*path: pathlib.Path*) → bool

class DomiasMIA(***kwargs: Any*)

Bases: *synthcity.metrics.eval_privacy.PrivacyEvaluator*

DOMIAS is a membership inference attacker model against synthetic data, that incorporates density estimation to detect generative model overfitting. That is it uses local overfitting to detect whether a data point was used to train the generative model or not.

Returns: A dictionary with a key for each of the *synthetic_sizes* values. For each *synthetic_sizes* value, the dictionary contains the keys:

- *MIA_performance* : accuracy and AUCROC for each attack
- *MIA_scores*: output scores for each attack

Reference: Boris van Breugel, Hao Sun, Zhaozhi Qian, Mihaela van der Schaar, AISTATS 2023. DOMIAS: Membership Inference Attacks against Synthetic Data through Overfitting Detection.

static direction() → str

evaluate(*X_gt:* synthcity.plugins.core.dataloader.DataLoader, *X_syn:* synthcity.plugins.core.dataloader.DataLoader, **args: Any*, ***kwargs: Any*) → Dict

evaluate_default(*X_gt:* synthcity.plugins.core.dataloader.DataLoader, *X_syn:* synthcity.plugins.core.dataloader.DataLoader, *X_train:* synthcity.plugins.core.dataloader.DataLoader, *X_ref_syn:* synthcity.plugins.core.dataloader.DataLoader, *reference_size: int*) → float

abstract evaluate_p_R(*synth_set: Union[*synthcity.plugins.core.dataloader.DataLoader*, Any]*, *synth_val_set: Union[*synthcity.plugins.core.dataloader.DataLoader*, Any]*, *reference_set: numpy.ndarray*, *X_test: numpy.ndarray*, *device: Any*) → Any

classmethod fqdn() → str

static name() → str

reduction() → Callable

static type() → str

use_cache(*path: pathlib.Path*) → bool

class DomiasMIABNAF(***kwargs: Any*)

Bases: *synthcity.metrics.eval_privacy.DomiasMIA*

static direction() → str

evaluate(*X_gt:* synthcity.plugins.core.dataloader.DataLoader, *X_syn:* synthcity.plugins.core.dataloader.DataLoader, **args: Any*, ***kwargs: Any*) → Dict

evaluate_default(*X_gt:* synthcity.plugins.core.dataloader.DataLoader, *X_syn:* synthcity.plugins.core.dataloader.DataLoader, *X_train:* synthcity.plugins.core.dataloader.DataLoader, *X_ref_syn:* synthcity.plugins.core.dataloader.DataLoader, *reference_size: int*) → float

evaluate_p_R(*synth_set: Union[*synthcity.plugins.core.dataloader.DataLoader*, Any]*, *synth_val_set: Union[*synthcity.plugins.core.dataloader.DataLoader*, Any]*, *reference_set: numpy.ndarray*, *X_test: numpy.ndarray*, *device: Any*) → Tuple[numpy.ndarray, numpy.ndarray]

classmethod fqdn() → str

static name() → str

reduction() → Callable

static type() → str

use_cache(*path: pathlib.Path*) → bool

class DomiasMIAKDE(***kwargs: Any*)

Bases: *synthcity.metrics.eval_privacy.DomiasMIA*

static direction() → str

evaluate(*X_gt:* synthcity.plugins.core.dataloader.DataLoader, *X_syn:* synthcity.plugins.core.dataloader.DataLoader, **args: Any*, ***kwargs: Any*) → Dict

evaluate_default(*X_gt:* synthcity.plugins.core.dataloader.DataLoader, *X_syn:* synthcity.plugins.core.dataloader.DataLoader, *X_train:* synthcity.plugins.core.dataloader.DataLoader, *X_ref_syn:* synthcity.plugins.core.dataloader.DataLoader, *reference_size: int*) → float

evaluate_p_R(*synth_set: Union[*synthcity.plugins.core.dataloader.DataLoader*, Any]*, *synth_val_set: Union[*synthcity.plugins.core.dataloader.DataLoader*, Any]*, *reference_set: numpy.ndarray*, *X_test: numpy.ndarray*, *device: Any*) → Tuple[numpy.ndarray, numpy.ndarray]

classmethod fqdn() → str

static name() → str

reduction() → Callable

static type() → str

use_cache(*path: pathlib.Path*) → bool

class DomiasMIAPrior(***kwargs: Any*)

Bases: *synthcity.metrics.eval_privacy.DomiasMIA*

static direction() → str

evaluate(*X_gt:* synthcity.plugins.core.dataloader.DataLoader, *X_syn:* synthcity.plugins.core.dataloader.DataLoader, **args: Any*, ***kwargs: Any*) → Dict

evaluate_default(*X_gt:* synthcity.plugins.core.dataloader.DataLoader, *X_syn:* synthcity.plugins.core.dataloader.DataLoader, *X_train:* synthcity.plugins.core.dataloader.DataLoader, *X_ref_syn:* synthcity.plugins.core.dataloader.DataLoader, *reference_size: int*) → float

evaluate_p_R(*synth_set: Union[*synthcity.plugins.core.dataloader.DataLoader*, Any]*, *synth_val_set: Union[*synthcity.plugins.core.dataloader.DataLoader*, Any]*, *reference_set: numpy.ndarray*, *X_test: numpy.ndarray*, *device: Any*) → Tuple[numpy.ndarray, numpy.ndarray]

classmethod fqdn() → str

static name() → str

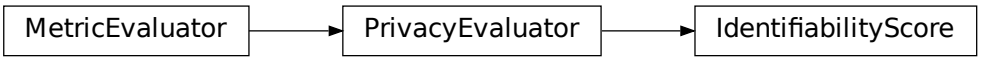
reduction() → Callable

static type() → str

use_cache(*path: pathlib.Path*) → bool

class IdentifiabilityScore(***kwargs: Any*)

Bases: *synthcity.metrics.eval_privacy.PrivacyEvaluator*



Returns the re-identification score on the real dataset from the synthetic dataset.

We estimate the risk of re-identifying any real data point using synthetic data. Intuitively, if the synthetic data are very close to the real data, the re-identification risk would be high. The precise formulation of the re-identification score is given in the reference below.

Reference: Jinsung Yoon, Lydia N. Drumright, Mihaela van der Schaar, "Anonymization through Data Synthesis using Generative Adversarial Networks (ADS-GAN): A harmonizing advancement for AI in medicine," IEEE Journal of Biomedical and Health Informatics (JBHI), 2019. Paper link: https://ieeexplore.ieee.org/document/9034117

static direction() → str

evaluate(*X_gt:* synthcity.plugins.core.dataloader.DataLoader, *X_syn:* synthcity.plugins.core.dataloader.DataLoader, **args: Any*, ***kwargs: Any*) → Dict

evaluate_default(*X_gt:* synthcity.plugins.core.dataloader.DataLoader, *X_syn:* synthcity.plugins.core.dataloader.DataLoader) → float

classmethod fqdn() → str

static name() → str

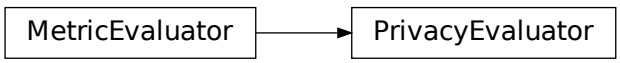
reduction() → Callable

static type() → str

use_cache(*path: pathlib.Path*) → bool

class PrivacyEvaluator(***kwargs: Any*)

Bases: synthcity.metrics.core.metric.MetricEvaluator



abstract static direction() → str

evaluate(*X_gt:* synthcity.plugins.core.dataloader.DataLoader, *X_syn:* synthcity.plugins.core.dataloader.DataLoader, **args: Any*, ***kwargs: Any*) → Dict

evaluate_default(*X_gt:* synthcity.plugins.core.dataloader.DataLoader, *X_syn:* synthcity.plugins.core.dataloader.DataLoader) → float

classmethod fqdn() → str

abstract static name() → str

reduction() → Callable

static type() → str

use_cache(*path: pathlib.Path*) → bool

class kAnonymization(***kwargs: Any*)

Bases: *synthcity.metrics.eval_privacy.PrivacyEvaluator*

Returns the k-anon ratio between the real data and the synthetic data. For each dataset, it is computed the value k which satisfies the k-anonymity rule: each record is similar to at least another k-1 other records on the potentially identifying variables.

static direction() → str

evaluate(*X_gt:* synthcity.plugins.core.dataloader.DataLoader, *X_syn:* synthcity.plugins.core.dataloader.DataLoader, **args: Any*, ***kwargs: Any*) → Dict

evaluate_data(*X:* synthcity.plugins.core.dataloader.DataLoader) → int

evaluate_default(*X_gt:* synthcity.plugins.core.dataloader.DataLoader, *X_syn:* synthcity.plugins.core.dataloader.DataLoader) → float

classmethod fqdn() → str

static name() → str

reduction() → Callable

static type() → str

use_cache(*path: pathlib.Path*) → bool

class kMap(***kwargs: Any*)

Bases: *synthcity.metrics.eval_privacy.PrivacyEvaluator*

Returns the minimum value k that satisfies the k-map rule.

The data satisfies k-map if every combination of values for the quasi-identifiers appears at least k times in the reidentification(synthetic) dataset.

static direction() → str

evaluate(*X_gt:* synthcity.plugins.core.dataloader.DataLoader, *X_syn:* synthcity.plugins.core.dataloader.DataLoader, **args: Any*, ***kwargs: Any*) → Dict

evaluate_default(*X_gt:* synthcity.plugins.core.dataloader.DataLoader, *X_syn:* synthcity.plugins.core.dataloader.DataLoader) → float

classmethod fqdn() → str

static name() → str

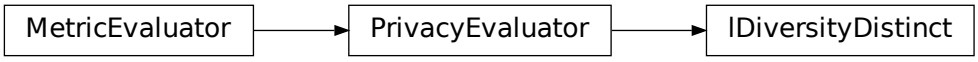
reduction() → Callable

static type() → str

use_cache(*path: pathlib.Path*) → bool

class lDiversityDistinct(***kwargs: Any*)

Bases: *synthcity.metrics.eval_privacy.PrivacyEvaluator*



Returns the distinct l-diversity ratio between the real data and the synthetic data.

For each dataset, it computes the minimum value l which satisfies the distinct l-diversity rule: every generalized block has to contain at least l different sensitive values.

We simulate a set of the cluster over the dataset, and we return the minimum length of unique sensitive values for any cluster.

static direction() → str

evaluate(*X_gt:* synthcity.plugins.core.dataloader.DataLoader, *X_syn:* synthcity.plugins.core.dataloader.DataLoader, **args: Any*, ***kwargs: Any*) → Dict

evaluate_data(*X:* synthcity.plugins.core.dataloader.DataLoader) → int

evaluate_default(*X_gt:* synthcity.plugins.core.dataloader.DataLoader, *X_syn:* synthcity.plugins.core.dataloader.DataLoader) → float

classmethod fqdn() → str

static name() → str

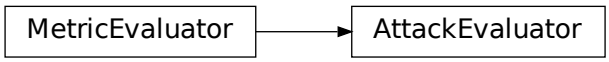
reduction() → Callable

static type() → str

use_cache(*path: pathlib.Path*) → bool

11.1.6 synthcity.metrics.eval_attacks module

class AttackEvaluator(***kwargs: Any*)
Bases: synthcity.metrics.core.metric.MetricEvaluator



Evaluating the risk of attribute inference attack.

This class evaluates the risk of a type of privacy attack, known as attribute inference attack. In this setting, the attacker has access to the synthetic dataset as well as partial information about the real data (quasi-identifiers). The attacker seeks to uncover the sensitive attributes of the real data using these two pieces of information.

abstract static direction() → str

abstract evaluate(*X_gt:* synthcity.plugins.core.dataloader.DataLoader, *X_syn:* synthcity.plugins.core.dataloader.DataLoader) → Dict

evaluate_default(*X_gt:* synthcity.plugins.core.dataloader.DataLoader, *X_syn:* synthcity.plugins.core.dataloader.DataLoader) → float

classmethod fqdn() → str

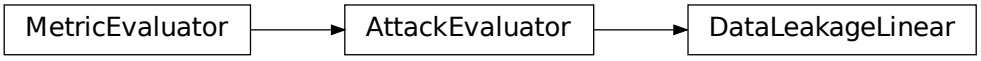
abstract static name() → str

reduction() → Callable

static type() → str

use_cache(*path: pathlib.Path*) → bool

class DataLeakageLinear(***kwargs: Any*)
Bases: *synthcity.metrics.eval_attacks.AttackEvaluator*



Data leakage test using a linear model

static direction() → str

evaluate(*X_gt:* synthcity.plugins.core.dataloader.DataLoader, *X_syn:* synthcity.plugins.core.dataloader.DataLoader) → Dict

evaluate_default(*X_gt:* synthcity.plugins.core.dataloader.DataLoader, *X_syn:* synthcity.plugins.core.dataloader.DataLoader) → float

classmethod fqdn() → str

static name() → str

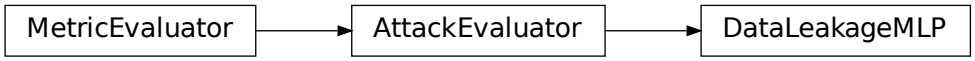
reduction() → Callable

static type() → str

use_cache(*path: pathlib.Path*) → bool

class DataLeakageMLP(***kwargs: Any*)

Bases: *synthcity.metrics.eval_attacks.AttackEvaluator*



Data leakage test using a neural net.

static direction() → str

evaluate(*X_gt:* synthcity.plugins.core.dataloader.DataLoader, *X_syn:* synthcity.plugins.core.dataloader.DataLoader) → Dict

evaluate_default(*X_gt:* synthcity.plugins.core.dataloader.DataLoader, *X_syn:* synthcity.plugins.core.dataloader.DataLoader) → float

classmethod fqdn() → str

static name() → str

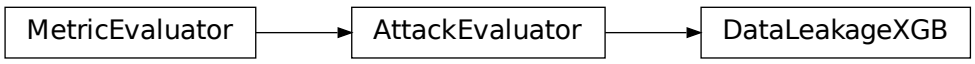
reduction() → Callable

static type() → str

use_cache(*path: pathlib.Path*) → bool

class DataLeakageXGB(***kwargs: Any*)

Bases: *synthcity.metrics.eval_attacks.AttackEvaluator*



Data leakage test using XGBoost

static direction() → str

evaluate(*X_gt:* synthcity.plugins.core.dataloader.DataLoader, *X_syn:* synthcity.plugins.core.dataloader.DataLoader) → Dict

evaluate_default(*X_gt:* synthcity.plugins.core.dataloader.DataLoader, *X_syn:* synthcity.plugins.core.dataloader.DataLoader) → float

classmethod fqdn() → str

static name() → str

reduction() → Callable

static type() → str

use_cache(*path: pathlib.Path*) → bool

11.1.7 synthcity.metrics.weighted_metrics module

class WeightedMetrics(*metrics: List[Tuple[str, str]]*, *weights: List[float]*, *task_type: str = 'classification'*, *random_state: int = 0*, *workspace: pathlib.Path = PosixPath('workspace')*)

Bases: object

direction() → str

evaluate(*X_gt: Union[*synthcity.plugins.core.dataloader.DataLoader*, pandas.core.frame.DataFrame]*, *X_syn: Union[*synthcity.plugins.core.dataloader.DataLoader*, pandas.core.frame.DataFrame]*) → float

11.1.8 synthcity.benchmark package

class Benchmarks

Bases: object

static evaluate(*tests: List[Tuple[str, str, dict]]*, *X:* synthcity.plugins.core.dataloader.DataLoader, *X_test: Optional[*synthcity.plugins.core.dataloader.DataLoader*] = None*, *metrics: Optional[Dict] = None*, *repeats: int = 3*, *synthetic_size: Optional[int] = None*, *synthetic_constraints: Optional[*synthcity.plugins.core.constraints.Constraints*] = None*, *synthetic_cache: bool = True*, *synthetic_reuse_if_exists: bool = True*, *augmented_reuse_if_exists: bool = True*, *task_type: str = 'classification'*, *workspace: pathlib.Path = PosixPath('workspace')*, *augmentation_rule: str = 'equal'*, *strict_augmentation: bool = False*, *ad_hoc_augment_vals: Optional[Dict] = None*, *use_metric_cache: bool = True*, ***generate_kwargs: Any*) → pandas.core.frame.DataFrame

Benchmark the performance of several algorithms.

Parameters

- **tests** – List[Tuple[str, str, dict]] Tuples of form (testname: str, plugin_name, str, plugin_args: dict)
- **X** – DataLoader The baseline dataset to learn
- **X_test** – Optional[DataLoader] Optional test dataset for evaluation. If None, X will be split in train/test datasets.
- **metrics** – List of metrics to test. By default, all metrics are evaluated. Full dictionary of metrics is: {

  'sanity': ['data_mismatch', 'common_rows_proportion', 'nearest_syn_neighbor_distance', 'close_values_probability', 'distant_values_probability'], 'stats': ['jensenshannon_dist', 'chi_squared_test', 'feature_corr', 'inv_kl_divergence', 'ks_test', 'max_mean_discrepancy', 'wasserstein_dist', 'prdc', 'alpha_precision', 'survival_km_distance'], 'performance': ['linear_model', 'mlp', 'xgb', 'feat_rank_distance'], 'detection': ['detection_xgb', 'detection_mlp', 'detection_gmm', 'detection_linear'], 'privacy': ['delta-presence', 'k-anonymization', 'k-map', 'distinct l-diversity',

'identifiability_score', 'DomiasMIA_BNAF', 'DomiasMIA_KDE', 'DomiasMIA_prior']

}

- **repeats** – Number of test repeats
- **synthetic_size** – int The size of the synthetic dataset. By default, it is len(X).
- **synthetic_constraints** – Optional constraints on the synthetic data. By default, it inherits the constraints from X.
- **synthetic_cache** – bool Enable experiment caching
- **synthetic_reuse_if_exists** – bool If the current synthetic dataset is cached, it will be reused for the experiments. Defaults to True.
- **augmented_reuse_if_exists** – bool If the current augmented dataset is cached, it will be reused for the experiments. Defaults to True.
- **task_type** – str The type of problem. Relevant for evaluating the downstream models with the correct metrics. Valid tasks are: "classification", "regression", "survival_analysis", "time_series", "time_series_survival".
- **workspace** – Path Path for caching experiments. Default: "workspace".
- **augmentation_rule** – str The rule used to achieve the desired proportion records with each value in the fairness column. Possible values are: 'equal', 'log', and 'ad-hoc'. Defaults to "equal".
- **strict_augmentation** – bool Flag to ensure that the condition for generating synthetic data is strictly met. Defaults to False.
- **ad_hoc_augment_vals** – Dict A dictionary containing the number of each class to augment the real data with. This is only required if using the rule="ad-hoc" option. Defaults to None.
- **use_metric_cache** – bool If the current metric has been previously run and is cached, it will be reused for the experiments. Defaults to True.
- **plugin_kwargs** – Optional kwargs for each algorithm. Example {"adsgan": {"n_iter": 10}},

static highlight(*results: Dict*) → None

static print(*results: Dict*, *only_comparatives: bool = True*) → None

print_score(*mean: pandas.core.series.Series*, *std: pandas.core.series.Series*) → pandas.core.series.Series

Submodules

synthcity.benchmark.utils module

augment_data(*X_train:* synthcity.plugins.core.dataloader.DataLoader, *augment_generator: Any*, *strict: bool = False*, *rule: typing_extensions.Literal[equal, log, ad - hoc] = 'equal'*, *ad_hoc_augment_vals: Optional[Dict[Any, int]] = None*, *synthetic_constraints: Optional[*synthcity.plugins.core.constraints.Constraints*] = None*, ***generate_kwargs: Any*) → *synthcity.plugins.core.dataloader.DataLoader*

Augment the real data with generated synthetic data

Parameters

- **X_train** (DataLoader) – The ground truth DataLoader to augment with synthetic data.
- **augment_generator** (*Any*) – The synthetic model to be used to generate the synthetic portion of the augmented dataset.
- **strict** (*bool, optional*) – Flag to ensure that the condition for generating synthetic data is strictly met. Defaults to False.
- **rule** (*Literal["equal", "log", "ad-hoc"*) – The rule used to achieve the desired proportion records with each value in the fairness column. Defaults to "equal".
- **ad_hoc_augment_vals** (*Dict[Union[int, str], int], optional*) – A dictionary containing the number of each class to augment the real data with. This is only required if using the rule="ad-hoc" option. Defaults to None.
- **synthetic_constraints** (*Optional[Constraints]*) – Constraints placed on the generation of the synthetic data. Defaults to None.

Returns The augmented dataset and labels.

Return type *DataLoader*

calculate_fair_aug_sample_size(*X_train: pandas.core.frame.DataFrame*, *fairness_column: Optional[str]*, *rule: typing_extensions.Literal[equal, log, ad - hoc]*, *ad_hoc_augment_vals: Optional[Dict[Any, int]] = None*) → Dict

Calculate how many samples to augment.

Parameters

- **X_train** (*pd.DataFrame*) – The real dataset to be augmented.
- **fairness_column** (*str*) – The column name of the column to test the fairness of a downstream model with respect to.
- **rule** (*Literal["equal", "log", "ad-hoc"]*) – The rule used to achieve the desired proportion records with each value in the fairness column. Defaults to "equal".
- **ad_hoc_augment_vals** (*Dict[Union[int, str], int], optional*) – A dictionary containing the number of each class to augment the real data with. If using rule="ad-hoc" this function returns ad_hoc_augment_vals, otherwise this parameter is ignored. Defaults to {}.

Returns A dictionary containing the number of each class to augment the real data with.

Return type Dict

get_json_serializable_kwargs(*kwargs: Dict*) → Dict

This function should take the kwargs for Benchmarks.evaluate and makes them serializable with json.dumps. Currently it only handles pathlib.Path -> str.

CHAPTER

TWELVE

ADVANCED TOPICS

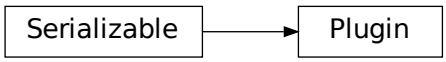
12.1 Advanced topics

12.1.1 Core components

synthcity.plugins.core.plugin module

class Plugin(*sampling_patience: int = 500*, *strict: bool = True*, *device: Any = device(type='cpu')*, *random_state: int = 0*, *workspace: pathlib.Path = PosixPath('workspace')*, *compress_dataset: bool = False*, *sampling_strategy: str = 'marginal'*)

Bases: *synthcity.plugins.core.serializable.Serializable*



Base class for all plugins.

Each derived class must implement the following methods: type() - a static method that returns the type of the plugin. e.g., debug, generative, bayesian, etc. name() - a static method that returns the name of the plugin. e.g., ctgan, random_noise, etc. hyperparameter_space() - a static method that returns the hyperparameters that can be tuned during AutoML. _fit() - internal method, called by *fit* on each training set. _generate() - internal method, called by *generate*.

If any method implementation is missing, the class constructor will fail.

Parameters

- **strict** – bool. Default = True If True, is raises an exception if the generated data is not following the requested constraints. If False, it returns only the rows that match the constraints.
- **workspace** – Path Path for caching intermediary results
- **compress_dataset** – bool. Default = False Drop redundant features before training the generator.
- **device** – PyTorch device: cpu or cuda.
- **random_state** – int Random seed

- **sampling_patience** – int. Max inference iterations to wait for the generated data to match the training schema.
- **sampling_strategy** – str Internal parameter for schema. marginal or uniform.

class Config

Bases: object

arbitrary_types_allowed = True

validate_assignment = True

fit(*X: Union[*synthcity.plugins.core.dataloader.DataLoader*, pandas.core.frame.DataFrame]*, **args: Any*, ***kwargs: Any*) → Any

Training method the synthetic data plugin.

Parameters

- **X** – DataLoader. The reference dataset.
- **cond** – Optional, Union[pd.DataFrame, pd.Series, np.ndarray] Optional Training Conditional. The training conditional can be used to control to output of some models, like GANs or VAEs. The content can be anything, as long as it maps to the training dataset X. Usage example:

```
>>> from sklearn.datasets import load_iris
>>> from synthcity.plugins.core.dataloader import␣
↪GenericDataLoader
>>> from synthcity.plugins.core.constraints import Constraints
>>>
>>> # Load in `test_plugin` the generative model of choice
>>> # ....
>>>
>>> X, y = load_iris(as_frame=True, return_X_y=True)
>>> X["target"] = y
>>>
>>> X = GenericDataLoader(X)
>>> test_plugin.fit(X, cond=y)
>>>
>>> count = 10
>>> X_gen = test_plugin.generate(count, cond=np.ones(count))
>>>
>>> # The Conditional only optimizes the output generation
>>> # for GANs and VAEs, but does NOT guarantee the samples
>>> # are only from that condition.
>>> # If you want to guarantee that output contains only
>>> # "target" == 1 samples, use Constraints.
>>>
>>> constraints = Constraints(
>>>     rules=[
>>>         ("target", "==", 1),
>>>     ]
>>> )
>>> X_gen = test_plugin.generate(count,
>>>         cond=np.ones(count),
>>>         constraints=constraints
>>>        )
>>> assert (X_gen["target"] == 1).all()
```

Returns self

classmethod fqdn() → str
The Fully-Qualified name of the plugin.

generate(*count: Optional[int] = None*, *constraints: Optional[*synthcity.plugins.core.constraints.Constraints*] = None*, *random_state: Optional[int] = None*, ***kwargs: Any*) → *synthcity.plugins.core.dataloader.DataLoader*
Synthetic data generation method.

Parameters

- **count** – optional int. The number of samples to generate. If None, it generated len(reference_dataset) samples.
- **cond** – Optional, Union[pd.DataFrame, pd.Series, np.ndarray]. Optional Generation Conditional. The conditional can be used only if the model was trained using a conditional too. If provided, it must have *count* length. Not all models support conditionals. The conditionals can be used in VAEs or GANs to speed-up the generation under some constraints. For model agnostic solutions, check out the *constraints* parameter.
- **constraints** – optional Constraints. Optional constraints to apply on the generated data. If none, the reference schema constraints are applied. The constraints are model agnostic, and will filter the output of the generative model. The constraints are a list of rules. Each rule is a tuple of the form (<feature>, <operation>, <value>).

  Valid Operations:

  - "<", "lt" : less than <value>
  - "<=", "le": less or equal with <value>
  - ">", "gt" : greater than <value>
  - ">=", "ge": greater or equal with <value>
  - "==", "eq": equal with <value>
  - "in": valid for categorical features, and <value> must be array. for example, ("target", "in", [0, 1])
  - "dtype": <value> can be a data type. For example, ("target", "dtype", "int")

  Usage example:

```
>>> from synthcity.plugins.core.constraints import Constraints
>>> constraints = Constraints(
>>>   rules=[
>>>             ("InterestingFeature", "==", 0),
>>>         ]
>>>     )
>>>
>>> syn_data = syn_model.generate(
        count=count,
        constraints=constraints
    ).dataframe()
>>>
>>> assert (syn_data["InterestingFeature"] == 0).all()
```

- **random_state** – optional int. Optional random seed to use.

Returns <count> synthetic samples

abstract static hyperparameter_space(***kwargs: Any*) → List[*synthcity.plugins.core.distribution.Distribution*]
Returns the hyperparameter space for the derived plugin.

static load(*buff: bytes*) → Any

static load_dict(*representation: dict*) → Any

abstract static name() → str
The name of the plugin.

plot(*plt: Any*, *X:* synthcity.plugins.core.dataloader.DataLoader, *count: Optional[int] = None*, *plots: list = ['marginal', 'associations', 'tsne']*, ***kwargs: Any*) → Any
Plot the real-synthetic distributions.

Parameters

- **plt** – output
- **X** – DataLoader. The reference dataset.

Returns self

classmethod sample_hyperparameters(**args: Any*, ***kwargs: Any*) → Dict[str, Any]
Sample value from the hyperparameter space for the current plugin.

classmethod sample_hyperparameters_optuna(*trial: Any*, **args: Any*, ***kwargs: Any*) → Dict[str, Any]

save() → bytes

save_dict() → dict

save_to_file(*path: pathlib.Path*) → bytes

schema() → *synthcity.plugins.core.schema.Schema*
The reference schema

schema_includes(*other: Union[*synthcity.plugins.core.dataloader.DataLoader*, pandas.core.frame.DataFrame]*) → bool
Helper method to test if the reference schema includes a Dataset

Parameters **other** – DataLoader. The dataset to test

Returns bool, if the schema includes the dataset or not.

training_schema() → *synthcity.plugins.core.schema.Schema*
The internal schema

abstract static type() → str
The type of the plugin.

static version() → str
API version

class PluginLoader(*plugins: list*, *expected_type: Type*, *categories: list*)
Bases: object

Plugin loading utility class. Used to load the plugins from the current folder.

add(*name: str*, *cls: Type*) → *synthcity.plugins.core.plugin.PluginLoader*
Add a new plugin

get(*name: str*, **args: Any*, ***kwargs: Any*) → Any
Create a new object from a plugin. :param name: str. The name of the plugin :param &args: :param **kwargs. Plugin specific arguments:

Returns The new object

get_type(*name: str*) → Type
Get the class type of a plugin. :param name: str. The name of the plugin

Returns The class of the plugin

list() → List[str]
Get all the available plugins.

load(*buff: bytes*) → Any
Load serialized plugin

reload() → *synthcity.plugins.core.plugin.PluginLoader*

types() → List[Type]
Get the loaded plugins types

synthcity.plugins.core.constraints module

class Constraints(*, *rules: list = []*)
Bases: pydantic.main.BaseModel

Constraints on data.

The Constraints class allows users to specify constraints on the features. Examples include the feature value range, allowed item set, and data type. These constraints can be used to filter out invalid values in synthetic datasets.

Constructor Args:

rules: List[Tuple] Each tuple in the list specifies a constraint on a feature. The tuple has the form of (feature, op, thresh), where feature is the feature name to apply constraint on, op takes values in [

"<", ">=", "<=", ">", "==", "lt", "le", "gt", "ge", "eq", "in", "dtype",

],

and thresh is the threshold or data type.

Config
alias of pydantic.config.BaseConfig

classmethod construct(*_fields_set: Optional[SetStr] = None*, ***values: Any*) → Model
Creates a new model setting __dict__ and __fields_set__ from trusted or pre-validated data. Default values are respected, but no other validation is performed. Behaves as if *Config.extra = 'allow'* was set since it adds all passed values

copy(*, *include: Optional[Union[AbstractSetIntStr, MappingIntStrAny]] = None*, *exclude: Optional[Union[AbstractSetIntStr, MappingIntStrAny]] = None*, *update: Optional[DictStrAny] = None*, *deep: bool = False*) → Model
Duplicate a model, optionally choose which fields to include, exclude and change.

Parameters

- **include** – fields to include in new model
- **exclude** – fields to exclude from new model, as with values this takes precedence over include
- **update** – values to change/add in the new model. Note: the data is not validated before creating the new model: you should trust this data
- **deep** – set to *True* to make a deep copy of the model

Returns new model instance

dict(*, *include: Optional[Union[AbstractSetIntStr, MappingIntStrAny]] = None*, *exclude: Optional[Union[AbstractSetIntStr, MappingIntStrAny]] = None*, *by_alias: bool = False*, *skip_defaults: Optional[bool] = None*, *exclude_unset: bool = False*, *exclude_defaults: bool = False*, *exclude_none: bool = False*) → DictStrAny

Generate a dictionary representation of the model, optionally specifying which fields to include or exclude.

extend(*other:* synthcity.plugins.core.constraints.Constraints) → *synthcity.plugins.core.constraints.Constraints*

Extend the local constraints with more constraints.

Parameters **other** – The new constraints to add.

Returns self with the updated constraints.

feature_constraints(*ref_feature: str*) → List

Get constraints for a given feature

Parameters **ref_feature** – str The name of the feature of interest.

Returns

[('le', 3.), ('gt', 1.)]

If ref_feature has no constraint, None will be returned.

Return type A list of tuples of (op, threshold) For example

feature_params(*feature: str*) → Tuple

Provide the parameters of Distribution from the Constraint

This is to be used with the constraint_to_distribution function in distribution module.

Parameters **feature** – str The name of the feature of interest.

Returns

str The type of inferred distribution from ("categorical", "float", "integer")

dist_args: Dict The arguments to the constructor of the Distribution.

Return type dist_template

features() → List

Return list of feature names in an undefined order

filter(*X: pandas.core.frame.DataFrame*) → pandas.core.frame.DataFrame

Apply the constraints to a DataFrame X.

Parameters **X** – DataFrame. The dataset to apply the constraints on.

Returns pandas.Index which matches all the constraints

classmethod from_orm(*obj: Any*) → Model

is_valid(*X: pandas.core.frame.DataFrame*) → bool
Checks if all the rows in X meet the constraints.

Parameters **X** – DataFrame. The dataset to apply the constraints on.

Returns True if all rows match the constraints, False otherwise

json(*, *include: Optional[Union[AbstractSetIntStr, MappingIntStrAny]] = None*, *exclude: Optional[Union[AbstractSetIntStr, MappingIntStrAny]] = None*, *by_alias: bool = False*, *skip_defaults: Optional[bool] = None*, *exclude_unset: bool = False*, *exclude_defaults: bool = False*, *exclude_none: bool = False*, *encoder: Optional[Callable[[Any], Any]] = None*, *models_as_dict: bool = True*, ***dumps_kwargs: Any*) → unicode
Generate a JSON representation of the model, *include* and *exclude* arguments as per *dict()*.

encoder is an optional function to supply as *default* to json.dumps(), other arguments as per *json.dumps()*.

match(*X: pandas.core.frame.DataFrame*) → pandas.core.frame.DataFrame
Apply the constraints to a DataFrame X and return the filtered dataset.

Parameters **X** – DataFrame. The dataset to apply the constraints on.

Returns The filtered Dataframe

classmethod parse_file(*path: Union[str, pathlib.Path]*, ***, *content_type: unicode = None*, *encoding: unicode = 'utf8'*, *proto: pydantic.parse.Protocol = None*, *allow_pickle: bool = False*) → Model

classmethod parse_obj(*obj: Any*) → Model

classmethod parse_raw(*b: Union[str, bytes]*, ***, *content_type: unicode = None*, *encoding: unicode = 'utf8'*, *proto: pydantic.parse.Protocol = None*, *allow_pickle: bool = False*) → Model

rules: list

classmethod schema(*by_alias: bool = True*, *ref_template: unicode = '#/definitions/{model}'*) → DictStrAny

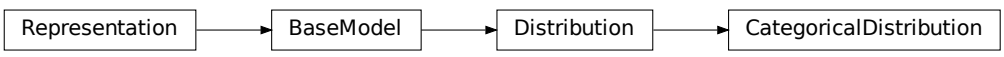
classmethod schema_json(*, *by_alias: bool = True*, *ref_template: unicode = '#/definitions/{model}'*, ***dumps_kwargs: Any*) → unicode

classmethod update_forward_refs(***localns: Any*) → None
Try to update ForwardRefs on fields based on this Model, globalns and localns.

classmethod validate(*value: Any*) → Model

synthcity.plugins.core.distribution module

class CategoricalDistribution(*, *name: str*, *data: Optional[pandas.core.series.Series] = None*, *random_state: int = 0*, *marginal_distribution: Optional[pandas.core.series.Series] = None*, *choices: list = []*)
Bases: *synthcity.plugins.core.distribution.Distribution*



class Config
Bases: object

arbitrary_types_allowed = True

as_constraint() → *synthcity.plugins.core.constraints.Constraints*
Convert the Distribution to a set of Constraints.

choices: list

classmethod construct(*_fields_set: Optional[SetStr] = None*, ***values: Any*) → Model
Creates a new model setting __dict__ and __fields_set__ from trusted or pre-validated data. Default values are respected, but no other validation is performed. Behaves as if *Config.extra = 'allow'* was set since it adds all passed values

copy(*, *include: Optional[Union[AbstractSetIntStr, MappingIntStrAny]] = None*, *exclude: Optional[Union[AbstractSetIntStr, MappingIntStrAny]] = None*, *update: Optional[DictStrAny] = None*, *deep: bool = False*) → Model
Duplicate a model, optionally choose which fields to include, exclude and change.

Parameters

- **include** – fields to include in new model
- **exclude** – fields to exclude from new model, as with values this takes precedence over include
- **update** – values to change/add in the new model. Note: the data is not validated before creating the new model: you should trust this data
- **deep** – set to *True* to make a deep copy of the model

Returns new model instance

data: Optional[pandas.core.series.Series]

dict(*, *include: Optional[Union[AbstractSetIntStr, MappingIntStrAny]] = None*, *exclude: Optional[Union[AbstractSetIntStr, MappingIntStrAny]] = None*, *by_alias: bool = False*, *skip_defaults: Optional[bool] = None*, *exclude_unset: bool = False*, *exclude_defaults: bool = False*, *exclude_none: bool = False*) → DictStrAny
Generate a dictionary representation of the model, optionally specifying which fields to include or exclude.

dtype() → str

classmethod from_orm(*obj: Any*) → Model

get() → List[Any]
Return the metadata of the Distribution.

has(*val: Any*) → bool
Test if a value is included in the Distribution.

includes(*other:* synthcity.plugins.core.distribution.Distribution) → bool
Test if another Distribution is included in the local one.

json(*, *include: Optional[Union[AbstractSetIntStr, MappingIntStrAny]] = None*, *exclude: Optional[Union[AbstractSetIntStr, MappingIntStrAny]] = None*, *by_alias: bool = False*, *skip_defaults: Optional[bool] = None*, *exclude_unset: bool = False*, *exclude_defaults: bool = False*, *exclude_none: bool = False*, *encoder: Optional[Callable[[Any], Any]] = None*, *models_as_dict: bool = True*, ***dumps_kwargs: Any*) → unicode
Generate a JSON representation of the model, *include* and *exclude* arguments as per *dict()*.

encoder is an optional function to supply as *default* to json.dumps(), other arguments as per *json.dumps()*.

marginal_distribution: Optional[pandas.core.series.Series]

marginal_probabilities() → Optional[List]

marginal_states() → Optional[List]

max() → Any
Get the max value of the distribution.

min() → Any
Get the min value of the distribution.

name: str

classmethod parse_file(*path: Union[str, pathlib.Path]*, *, *content_type: unicode = None*, *encoding: unicode = 'utf8'*, *proto: pydantic.parse.Protocol = None*, *allow_pickle: bool = False*) → Model

classmethod parse_obj(*obj: Any*) → Model

classmethod parse_raw(*b: Union[str, bytes]*, *, *content_type: unicode = None*, *encoding: unicode = 'utf8'*, *proto: pydantic.parse.Protocol = None*, *allow_pickle: bool = False*) → Model

random_state: int

sample(*count: int = 1*) → Any
Sample a value from the Distribution.

sample_marginal(*count: int = 1*) → Any

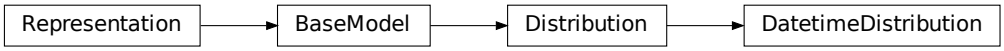
classmethod schema(*by_alias: bool = True*, *ref_template: unicode = '#/definitions/{model}'*) → DictStrAny

classmethod schema_json(*, *by_alias: bool = True*, *ref_template: unicode = '#/definitions/{model}'*, ***dumps_kwargs: Any*) → unicode

classmethod update_forward_refs(***localns: Any*) → None
Try to update ForwardRefs on fields based on this Model, globalns and localns.

classmethod validate(*value: Any*) → Model

class DatetimeDistribution(*, *name: str*, *data: Optional[pandas.core.series.Series] = None*, *random_state: int = 0*, *marginal_distribution: Optional[pandas.core.series.Series] = None*, *low: datetime.datetime = datetime.datetime(1970, 1, 1, 0, 0)*, *high: datetime.datetime = datetime.datetime(2024, 3, 11, 11, 44, 28, 414668)*, *step: datetime.timedelta = datetime.timedelta(microseconds=1)*, *offset: datetime.timedelta = datetime.timedelta(seconds=120)*)
Bases: *synthcity.plugins.core.distribution.Distribution*



class Config
Bases: object

arbitrary_types_allowed = True

as_constraint() → *synthcity.plugins.core.constraints.Constraints*
Convert the Distribution to a set of Constraints.

classmethod construct(*_fields_set: Optional[SetStr] = None*, ***values: Any*) → Model
Creates a new model setting __dict__ and __fields_set__ from trusted or pre-validated data. Default values are respected, but no other validation is performed. Behaves as if *Config.extra = 'allow'* was set since it adds all passed values

copy(***, *include: Optional[Union[AbstractSetIntStr, MappingIntStrAny]] = None*, *exclude: Optional[Union[AbstractSetIntStr, MappingIntStrAny]] = None*, *update: Optional[DictStrAny] = None*, *deep: bool = False*) → Model
Duplicate a model, optionally choose which fields to include, exclude and change.

Parameters

- **include** – fields to include in new model
- **exclude** – fields to exclude from new model, as with values this takes precedence over include
- **update** – values to change/add in the new model. Note: the data is not validated before creating the new model: you should trust this data
- **deep** – set to *True* to make a deep copy of the model

Returns new model instance

data: Optional[pandas.core.series.Series]

dict(***, *include: Optional[Union[AbstractSetIntStr, MappingIntStrAny]] = None*, *exclude: Optional[Union[AbstractSetIntStr, MappingIntStrAny]] = None*, *by_alias: bool = False*, *skip_defaults: Optional[bool] = None*, *exclude_unset: bool = False*, *exclude_defaults: bool = False*, *exclude_none: bool = False*) → DictStrAny
Generate a dictionary representation of the model, optionally specifying which fields to include or exclude.

dtype() → str

classmethod from_orm(*obj: Any*) → Model

get() → List[Any]
Return the metadata of the Distribution.

has(*val: datetime.datetime*) → bool
Test if a value is included in the Distribution.

high: datetime.datetime

includes(*other:* synthcity.plugins.core.distribution.Distribution) → bool
Test if another Distribution is included in the local one.

json(***, *include: Optional[Union[AbstractSetIntStr, MappingIntStrAny]] = None*, *exclude: Optional[Union[AbstractSetIntStr, MappingIntStrAny]] = None*, *by_alias: bool = False*, *skip_defaults: Optional[bool] = None*, *exclude_unset: bool = False*, *exclude_defaults: bool = False*, *exclude_none: bool = False*, *encoder: Optional[Callable[[Any], Any]] = None*, *models_as_dict: bool = True*, ***dumps_kwargs: Any*) → unicode
Generate a JSON representation of the model, *include* and *exclude* arguments as per *dict()*.

encoder is an optional function to supply as *default* to json.dumps(), other arguments as per *json.dumps()*.

low: datetime.datetime

marginal_distribution: Optional[pandas.core.series.Series]

marginal_probabilities() → Optional[List]

marginal_states() → Optional[List]

max() → Any
Get the max value of the distribution.

min() → Any
Get the min value of the distribution.

name: str

offset: datetime.timedelta

classmethod parse_file(*path: Union[str, pathlib.Path]*, *, *content_type: unicode = None*, *encoding: unicode = 'utf8'*, *proto: pydantic.parse.Protocol = None*, *allow_pickle: bool = False*) → Model

classmethod parse_obj(*obj: Any*) → Model

classmethod parse_raw(*b: Union[str, bytes]*, *, *content_type: unicode = None*, *encoding: unicode = 'utf8'*, *proto: pydantic.parse.Protocol = None*, *allow_pickle: bool = False*) → Model

random_state: int

sample(*count: int = 1*) → Any
Sample a value from the Distribution.

sample_marginal(*count: int = 1*) → Any

classmethod schema(*by_alias: bool = True*, *ref_template: unicode = '#/definitions/{model}'*) → DictStrAny

classmethod schema_json(*, *by_alias: bool = True*, *ref_template: unicode = '#/definitions/{model}'*, ***dumps_kwargs: Any*) → unicode

step: datetime.timedelta

classmethod update_forward_refs(***localns: Any*) → None
Try to update ForwardRefs on fields based on this Model, globalns and localns.

classmethod validate(*value: Any*) → Model

class Distribution(*, *name: str*, *data: Optional[pandas.core.series.Series] = None*, *random_state: int = 0*, *marginal_distribution: Optional[pandas.core.series.Series] = None*)
Bases: pydantic.main.BaseModel

Base class of all Distributions.

The Distribution class characterizes the **empirical** marginal distribution of the feature. Each derived class must implement the following methods:

get() - Return the metadata of the Distribution. sample() - Sample a value from the Distribution. includes() - Test if another Distribution is included in the local one. has() - Test if a value is included in the support of the Distribution. as_constraint() - Convert the Distribution to a set of Constraints.

min() - Return the minimum of the support. max() - Return the maximum of the support. __eq__() - Testing equality of two Distributions. dtype() - Return the data type

Examples of derived classes include CategoricalDistribution, FloatDistribution, and IntegerDistribution.

class Config
Bases: object

arbitrary_types_allowed = True

abstract as_constraint() → *synthcity.plugins.core.constraints.Constraints*
Convert the Distribution to a set of Constraints.

classmethod construct(*_fields_set: Optional[SetStr] = None*, ***values: Any*) → Model
Creates a new model setting __dict__ and __fields_set__ from trusted or pre-validated data. Default values are respected, but no other validation is performed. Behaves as if *Config.extra = 'allow'* was set since it adds all passed values

copy(***, *include: Optional[Union[AbstractSetIntStr, MappingIntStrAny]] = None*, *exclude: Optional[Union[AbstractSetIntStr, MappingIntStrAny]] = None*, *update: Optional[DictStrAny] = None*, *deep: bool = False*) → Model
Duplicate a model, optionally choose which fields to include, exclude and change.

Parameters

- **include** – fields to include in new model
- **exclude** – fields to exclude from new model, as with values this takes precedence over include
- **update** – values to change/add in the new model. Note: the data is not validated before creating the new model: you should trust this data
- **deep** – set to *True* to make a deep copy of the model

Returns new model instance

data: Optional[pandas.core.series.Series]

dict(***, *include: Optional[Union[AbstractSetIntStr, MappingIntStrAny]] = None*, *exclude: Optional[Union[AbstractSetIntStr, MappingIntStrAny]] = None*, *by_alias: bool = False*, *skip_defaults: Optional[bool] = None*, *exclude_unset: bool = False*, *exclude_defaults: bool = False*, *exclude_none: bool = False*) → DictStrAny
Generate a dictionary representation of the model, optionally specifying which fields to include or exclude.

abstract dtype() → str

classmethod from_orm(*obj: Any*) → Model

abstract get() → List[Any]
Return the metadata of the Distribution.

abstract has(*val: Any*) → bool
Test if a value is included in the Distribution.

abstract includes(*other:* synthcity.plugins.core.distribution.Distribution) → bool
Test if another Distribution is included in the local one.

json(***, *include: Optional[Union[AbstractSetIntStr, MappingIntStrAny]] = None*, *exclude: Optional[Union[AbstractSetIntStr, MappingIntStrAny]] = None*, *by_alias: bool = False*, *skip_defaults: Optional[bool] = None*, *exclude_unset: bool = False*, *exclude_defaults: bool = False*, *exclude_none: bool = False*, *encoder: Optional[Callable[[Any], Any]] = None*, *models_as_dict: bool = True*, ***dumps_kwargs: Any*) → unicode
Generate a JSON representation of the model, *include* and *exclude* arguments as per *dict()*.

encoder is an optional function to supply as *default* to json.dumps(), other arguments as per *json.dumps()*.

marginal_distribution: Optional[pandas.core.series.Series]

marginal_probabilities() → Optional[List]

marginal_states() → Optional[List]

abstract max() → Any
Get the max value of the distribution.

abstract min() → Any
Get the min value of the distribution.

name: str

classmethod parse_file(*path: Union[str, pathlib.Path]*, *, *content_type: unicode = None*, *encoding: unicode = 'utf8'*, *proto: pydantic.parse.Protocol = None*, *allow_pickle: bool = False*) → Model

classmethod parse_obj(*obj: Any*) → Model

classmethod parse_raw(*b: Union[str, bytes]*, *, *content_type: unicode = None*, *encoding: unicode = 'utf8'*, *proto: pydantic.parse.Protocol = None*, *allow_pickle: bool = False*) → Model

random_state: int

abstract sample(*count: int = 1*) → Any
Sample a value from the Distribution.

sample_marginal(*count: int = 1*) → Any

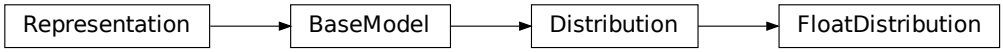
classmethod schema(*by_alias: bool = True*, *ref_template: unicode = '#/definitions/{model}'*) → DictStrAny

classmethod schema_json(*, *by_alias: bool = True*, *ref_template: unicode = '#/definitions/{model}'*, ***dumps_kwargs: Any*) → unicode

classmethod update_forward_refs(***localns: Any*) → None
Try to update ForwardRefs on fields based on this Model, globalns and localns.

classmethod validate(*value: Any*) → Model

class FloatDistribution(*, *name: str*, *data: Optional[pandas.core.series.Series] = None*, *random_state: int = 0*, *marginal_distribution: Optional[pandas.core.series.Series] = None*, *low: float = -1.7976931348623157e+308*, *high: float = 1.7976931348623157e+308*)
Bases: *synthcity.plugins.core.distribution.Distribution*



class Config
Bases: object

arbitrary_types_allowed = True

as_constraint() → *synthcity.plugins.core.constraints.Constraints*
Convert the Distribution to a set of Constraints.

classmethod construct(*_fields_set: Optional[SetStr] = None*, ***values: Any*) → Model
Creates a new model setting __dict__ and __fields_set__ from trusted or pre-validated data. Default values are respected, but no other validation is performed. Behaves as if *Config.extra = 'allow'* was set since it adds all passed values

copy(***, *include: Optional[Union[AbstractSetIntStr, MappingIntStrAny]] = None*, *exclude: Optional[Union[AbstractSetIntStr, MappingIntStrAny]] = None*, *update: Optional[DictStrAny] = None*, *deep: bool = False*) → Model
Duplicate a model, optionally choose which fields to include, exclude and change.

Parameters

- **include** – fields to include in new model
- **exclude** – fields to exclude from new model, as with values this takes precedence over include
- **update** – values to change/add in the new model. Note: the data is not validated before creating the new model: you should trust this data
- **deep** – set to *True* to make a deep copy of the model

Returns new model instance

data: Optional[pandas.core.series.Series]

dict(***, *include: Optional[Union[AbstractSetIntStr, MappingIntStrAny]] = None*, *exclude: Optional[Union[AbstractSetIntStr, MappingIntStrAny]] = None*, *by_alias: bool = False*, *skip_defaults: Optional[bool] = None*, *exclude_unset: bool = False*, *exclude_defaults: bool = False*, *exclude_none: bool = False*) → DictStrAny
Generate a dictionary representation of the model, optionally specifying which fields to include or exclude.

dtype() → str

classmethod from_orm(*obj: Any*) → Model

get() → List[Any]
Return the metadata of the Distribution.

has(*val: Any*) → bool
Test if a value is included in the Distribution.

high: float

includes(*other:* synthcity.plugins.core.distribution.Distribution) → bool
Test if another Distribution is included in the local one.

json(***, *include: Optional[Union[AbstractSetIntStr, MappingIntStrAny]] = None*, *exclude: Optional[Union[AbstractSetIntStr, MappingIntStrAny]] = None*, *by_alias: bool = False*, *skip_defaults: Optional[bool] = None*, *exclude_unset: bool = False*, *exclude_defaults: bool = False*, *exclude_none: bool = False*, *encoder: Optional[Callable[[Any], Any]] = None*, *models_as_dict: bool = True*, ***dumps_kwargs: Any*) → unicode
Generate a JSON representation of the model, *include* and *exclude* arguments as per *dict()*.

encoder is an optional function to supply as *default* to json.dumps(), other arguments as per *json.dumps()*.

low: float

marginal_distribution: Optional[pandas.core.series.Series]

marginal_probabilities() → Optional[List]

marginal_states() → Optional[List]

max() → Any
Get the max value of the distribution.

min() → Any
Get the min value of the distribution.

name: str

classmethod parse_file(*path: Union[str, pathlib.Path]*, *, *content_type: unicode = None*, *encoding: unicode = 'utf8'*, *proto: pydantic.parse.Protocol = None*, *allow_pickle: bool = False*) → Model

classmethod parse_obj(*obj: Any*) → Model

classmethod parse_raw(*b: Union[str, bytes]*, *, *content_type: unicode = None*, *encoding: unicode = 'utf8'*, *proto: pydantic.parse.Protocol = None*, *allow_pickle: bool = False*) → Model

random_state: int

sample(*count: int = 1*) → Any
Sample a value from the Distribution.

sample_marginal(*count: int = 1*) → Any

classmethod schema(*by_alias: bool = True*, *ref_template: unicode = '#/definitions/{model}'*) → DictStrAny

classmethod schema_json(*, *by_alias: bool = True*, *ref_template: unicode = '#/definitions/{model}'*, ***dumps_kwargs: Any*) → unicode

classmethod update_forward_refs(***localns: Any*) → None
Try to update ForwardRefs on fields based on this Model, globalns and localns.

classmethod validate(*value: Any*) → Model

class IntLogDistribution(*, *name: str*, *data: Optional[pandas.core.series.Series] = None*, *random_state: int = 0*, *marginal_distribution: Optional[pandas.core.series.Series] = None*, *low: int = 1*, *high: int = 9223372036854775807*, *step: int = 1*)
Bases: *synthcity.plugins.core.distribution.IntegerDistribution*

class Config
Bases: object

arbitrary_types_allowed = True

as_constraint() → *synthcity.plugins.core.constraints.Constraints*
Convert the Distribution to a set of Constraints.

classmethod construct(*_fields_set: Optional[SetStr] = None*, ***values: Any*) → Model
Creates a new model setting __dict__ and __fields_set__ from trusted or pre-validated data. Default values are respected, but no other validation is performed. Behaves as if *Config.extra = 'allow'* was set since it adds all passed values

copy(*, *include: Optional[Union[AbstractSetIntStr, MappingIntStrAny]] = None*, *exclude: Optional[Union[AbstractSetIntStr, MappingIntStrAny]] = None*, *update: Optional[DictStrAny] = None*, *deep: bool = False*) → Model
Duplicate a model, optionally choose which fields to include, exclude and change.

Parameters

- **include** – fields to include in new model
- **exclude** – fields to exclude from new model, as with values this takes precedence over include

- **update** – values to change/add in the new model. Note: the data is not validated before creating the new model: you should trust this data
- **deep** – set to *True* to make a deep copy of the model

Returns new model instance

data: Optional[pandas.core.series.Series]

dict(*, *include: Optional[Union[AbstractSetIntStr, MappingIntStrAny]] = None*, *exclude: Optional[Union[AbstractSetIntStr, MappingIntStrAny]] = None*, *by_alias: bool = False*, *skip_defaults: Optional[bool] = None*, *exclude_unset: bool = False*, *exclude_defaults: bool = False*, *exclude_none: bool = False*) → DictStrAny
Generate a dictionary representation of the model, optionally specifying which fields to include or exclude.

dtype() → str

classmethod from_orm(*obj: Any*) → Model

get() → List[Any]
Return the metadata of the Distribution.

has(*val: Any*) → bool
Test if a value is included in the Distribution.

high: int

includes(*other:* synthcity.plugins.core.distribution.Distribution) → bool
Test if another Distribution is included in the local one.

json(*, *include: Optional[Union[AbstractSetIntStr, MappingIntStrAny]] = None*, *exclude: Optional[Union[AbstractSetIntStr, MappingIntStrAny]] = None*, *by_alias: bool = False*, *skip_defaults: Optional[bool] = None*, *exclude_unset: bool = False*, *exclude_defaults: bool = False*, *exclude_none: bool = False*, *encoder: Optional[Callable[[Any], Any]] = None*, *models_as_dict: bool = True*, ***dumps_kwargs: Any*) → unicode
Generate a JSON representation of the model, *include* and *exclude* arguments as per *dict()*.

encoder is an optional function to supply as *default* to json.dumps(), other arguments as per *json.dumps()*.

low: int

marginal_distribution: Optional[pandas.core.series.Series]

marginal_probabilities() → Optional[List]

marginal_states() → Optional[List]

max() → Any
Get the max value of the distribution.

min() → Any
Get the min value of the distribution.

name: str

classmethod parse_file(*path: Union[str, pathlib.Path]*, *, *content_type: unicode = None*, *encoding: unicode = 'utf8'*, *proto: pydantic.parse.Protocol = None*, *allow_pickle: bool = False*) → Model

classmethod parse_obj(*obj: Any*) → Model

classmethod parse_raw(*b: Union[str, bytes]*, *, *content_type: unicode = None*, *encoding: unicode = 'utf8'*, *proto: pydantic.parse.Protocol = None*, *allow_pickle: bool = False*) → Model

random_state: int

sample(*count: int = 1*) → Any
Sample a value from the Distribution.

sample_marginal(*count: int = 1*) → Any

classmethod schema(*by_alias: bool = True*, *ref_template: unicode = '#/definitions/{model}'*) → DictStrAny

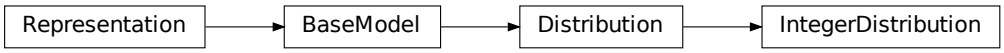
classmethod schema_json(*, *by_alias: bool = True*, *ref_template: unicode = '#/definitions/{model}'*, ***dumps_kwargs: Any*) → unicode

step: int

classmethod update_forward_refs(***localns: Any*) → None
Try to update ForwardRefs on fields based on this Model, globalns and localns.

classmethod validate(*value: Any*) → Model

class IntegerDistribution(*, *name: str*, *data: Optional[pandas.core.series.Series] = None*, *random_state: int = 0*, *marginal_distribution: Optional[pandas.core.series.Series] = None*, *low: int = -9223372036854775808*, *high: int = 9223372036854775807*, *step: int = 1*)
Bases: *synthcity.plugins.core.distribution.Distribution*



class Config
Bases: object

arbitrary_types_allowed = True

as_constraint() → *synthcity.plugins.core.constraints.Constraints*
Convert the Distribution to a set of Constraints.

classmethod construct(*_fields_set: Optional[SetStr] = None*, ***values: Any*) → Model
Creates a new model setting __dict__ and __fields_set__ from trusted or pre-validated data. Default values are respected, but no other validation is performed. Behaves as if *Config.extra = 'allow'* was set since it adds all passed values

copy(*, *include: Optional[Union[AbstractSetIntStr, MappingIntStrAny]] = None*, *exclude: Optional[Union[AbstractSetIntStr, MappingIntStrAny]] = None*, *update: Optional[DictStrAny] = None*, *deep: bool = False*) → Model
Duplicate a model, optionally choose which fields to include, exclude and change.

Parameters

- **include** – fields to include in new model
- **exclude** – fields to exclude from new model, as with values this takes precedence over include
- **update** – values to change/add in the new model. Note: the data is not validated before creating the new model: you should trust this data
- **deep** – set to *True* to make a deep copy of the model

Returns new model instance

data: Optional[pandas.core.series.Series]

dict(*, *include: Optional[Union[AbstractSetIntStr, MappingIntStrAny]] = None*, *exclude: Optional[Union[AbstractSetIntStr, MappingIntStrAny]] = None*, *by_alias: bool = False*, *skip_defaults: Optional[bool] = None*, *exclude_unset: bool = False*, *exclude_defaults: bool = False*, *exclude_none: bool = False*) → DictStrAny
Generate a dictionary representation of the model, optionally specifying which fields to include or exclude.

dtype() → str

classmethod from_orm(*obj: Any*) → Model

get() → List[Any]
Return the metadata of the Distribution.

has(*val: Any*) → bool
Test if a value is included in the Distribution.

high: int

includes(*other:* synthcity.plugins.core.distribution.Distribution) → bool
Test if another Distribution is included in the local one.

json(*, *include: Optional[Union[AbstractSetIntStr, MappingIntStrAny]] = None*, *exclude: Optional[Union[AbstractSetIntStr, MappingIntStrAny]] = None*, *by_alias: bool = False*, *skip_defaults: Optional[bool] = None*, *exclude_unset: bool = False*, *exclude_defaults: bool = False*, *exclude_none: bool = False*, *encoder: Optional[Callable[[Any], Any]] = None*, *models_as_dict: bool = True*, ***dumps_kwargs: Any*) → unicode
Generate a JSON representation of the model, *include* and *exclude* arguments as per *dict()*.

encoder is an optional function to supply as *default* to json.dumps(), other arguments as per *json.dumps()*.

low: int

marginal_distribution: Optional[pandas.core.series.Series]

marginal_probabilities() → Optional[List]

marginal_states() → Optional[List]

max() → Any
Get the max value of the distribution.

min() → Any
Get the min value of the distribution.

name: str

classmethod parse_file(*path: Union[str, pathlib.Path]*, *, *content_type: unicode = None*, *encoding: unicode = 'utf8'*, *proto: pydantic.parse.Protocol = None*, *allow_pickle: bool = False*) → Model

classmethod parse_obj(*obj: Any*) → Model

classmethod parse_raw(*b: Union[str, bytes]*, *, *content_type: unicode = None*, *encoding: unicode = 'utf8'*, *proto: pydantic.parse.Protocol = None*, *allow_pickle: bool = False*) → Model

random_state: int

sample(*count: int = 1*) → Any
Sample a value from the Distribution.

sample_marginal(*count: int = 1*) → Any

classmethod schema(*by_alias: bool = True*, *ref_template: unicode = '#/definitions/{model}'*) → DictStrAny

classmethod schema_json(*, *by_alias: bool = True*, *ref_template: unicode = '#/definitions/{model}'*, ***dumps_kwargs: Any*) → unicode

step: int

classmethod update_forward_refs(***localns: Any*) → None
Try to update ForwardRefs on fields based on this Model, globalns and localns.

classmethod validate(*value: Any*) → Model

class LogDistribution(*, *name: str*, *data: Optional[pandas.core.series.Series] = None*, *random_state: int = 0*, *marginal_distribution: Optional[pandas.core.series.Series] = None*, *low: float = 2.2250738585072014e-308*, *high: float = 1.7976931348623157e+308*)
Bases: *synthcity.plugins.core.distribution.FloatDistribution*

class Config
Bases: object

arbitrary_types_allowed = True

as_constraint() → *synthcity.plugins.core.constraints.Constraints*
Convert the Distribution to a set of Constraints.

classmethod construct(*_fields_set: Optional[SetStr] = None*, ***values: Any*) → Model
Creates a new model setting __dict__ and __fields_set__ from trusted or pre-validated data. Default values are respected, but no other validation is performed. Behaves as if *Config.extra = 'allow'* was set since it adds all passed values

copy(*, *include: Optional[Union[AbstractSetIntStr, MappingIntStrAny]] = None*, *exclude: Optional[Union[AbstractSetIntStr, MappingIntStrAny]] = None*, *update: Optional[DictStrAny] = None*, *deep: bool = False*) → Model
Duplicate a model, optionally choose which fields to include, exclude and change.

Parameters

- **include** – fields to include in new model
- **exclude** – fields to exclude from new model, as with values this takes precedence over include
- **update** – values to change/add in the new model. Note: the data is not validated before creating the new model: you should trust this data
- **deep** – set to *True* to make a deep copy of the model

Returns new model instance

data: Optional[pandas.core.series.Series]

dict(*, *include: Optional[Union[AbstractSetIntStr, MappingIntStrAny]] = None*, *exclude: Optional[Union[AbstractSetIntStr, MappingIntStrAny]] = None*, *by_alias: bool = False*, *skip_defaults: Optional[bool] = None*, *exclude_unset: bool = False*, *exclude_defaults: bool = False*, *exclude_none: bool = False*) → DictStrAny
Generate a dictionary representation of the model, optionally specifying which fields to include or exclude.

dtype() → str

classmethod from_orm(*obj: Any*) → Model

get() → List[Any]
Return the metadata of the Distribution.

has(*val: Any*) → bool
Test if a value is included in the Distribution.

high: float

includes(*other:* synthcity.plugins.core.distribution.Distribution) → bool
Test if another Distribution is included in the local one.

json(**, include: Optional[Union[AbstractSetIntStr, MappingIntStrAny]] = None, exclude: Optional[Union[AbstractSetIntStr, MappingIntStrAny]] = None, by_alias: bool = False, skip_defaults: Optional[bool] = None, exclude_unset: bool = False, exclude_defaults: bool = False, exclude_none: bool = False, encoder: Optional[Callable[[Any], Any]] = None, models_as_dict: bool = True, **dumps_kwargs: Any*) → unicode
Generate a JSON representation of the model, *include* and *exclude* arguments as per *dict()*.

encoder is an optional function to supply as *default* to json.dumps(), other arguments as per *json.dumps()*.

low: float

marginal_distribution: Optional[pandas.core.series.Series]

marginal_probabilities() → Optional[List]

marginal_states() → Optional[List]

max() → Any
Get the max value of the distribution.

min() → Any
Get the min value of the distribution.

name: str

classmethod parse_file(*path: Union[str, pathlib.Path], *, content_type: unicode = None, encoding: unicode = 'utf8', proto: pydantic.parse.Protocol = None, allow_pickle: bool = False*) → Model

classmethod parse_obj(*obj: Any*) → Model

classmethod parse_raw(*b: Union[str, bytes], *, content_type: unicode = None, encoding: unicode = 'utf8', proto: pydantic.parse.Protocol = None, allow_pickle: bool = False*) → Model

random_state: int

sample(*count: int = 1*) → Any
Sample a value from the Distribution.

sample_marginal(*count: int = 1*) → Any

classmethod schema(*by_alias: bool = True, ref_template: unicode = '#/definitions/{model}'*) → DictStrAny

classmethod schema_json(**, by_alias: bool = True, ref_template: unicode = '#/definitions/{model}', **dumps_kwargs: Any*) → unicode

classmethod update_forward_refs(***localns: Any*) → None
Try to update ForwardRefs on fields based on this Model, globalns and localns.

classmethod validate(*value: Any*) → Model

constraint_to_distribution(*constraints:* synthcity.plugins.core.constraints.Constraints, *feature: str*) → *synthcity.plugins.core.distribution.Distribution*
Infer Distribution from Constraints.

Parameters

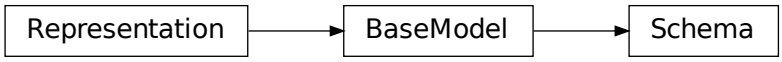
- **constraints** – Constraints The Constraints on features.
- **feature** – str The name of the feature in question.

Returns The inferred Distribution.

synthcity.plugins.core.schema module

class Schema(*, *sampling_strategy: str = 'marginal'*, *protected_cols: List[str] = ['seq_id']*, *random_state: int = 0*, *data: Any = None*, *domain: Dict = {}*)

Bases: pydantic.main.BaseModel



Utility class for defining the schema of a Dataset.

Constructor Args:

domain: Dict A dictionary of feature_name: Distribution.

sampling_strategy: str Taking value of "marginal" (default) or "uniform" (for debugging).

protected_cols: List[str] List of columns that are exempt from distributional constraints (e.g. ID column)

random_state: int Random seed (default 0)

data: Any (Optional) the data set

Config

alias of pydantic.config.BaseConfig

adapt_dtypes(*X: pandas.core.frame.DataFrame*) → pandas.core.frame.DataFrame

Applying the data type to a new data frame

Parameters **X** – pd.DataFrame A new data frame to be adapted.

Returns A data frame whose data types are coerced to be the same with the Schema. If the data frame contains new features, these will be retained as is.

as_constraints() → *synthcity.plugins.core.constraints.Constraints*

Convert the schema to a list of Constraints.

classmethod construct(*_fields_set: Optional[SetStr] = None*, ***values: Any*) → Model

Creates a new model setting __dict__ and __fields_set__ from trusted or pre-validated data. Default values are respected, but no other validation is performed. Behaves as if *Config.extra = 'allow'* was set since it adds all passed values

copy(*, *include: Optional[Union[AbstractSetIntStr, MappingIntStrAny]] = None*, *exclude: Optional[Union[AbstractSetIntStr, MappingIntStrAny]] = None*, *update: Optional[DictStrAny] = None*, *deep: bool = False*) → Model

Duplicate a model, optionally choose which fields to include, exclude and change.

Parameters

- **include** – fields to include in new model

- **exclude** – fields to exclude from new model, as with values this takes precedence over include
- **update** – values to change/add in the new model. Note: the data is not validated before creating the new model: you should trust this data
- **deep** – set to *True* to make a deep copy of the model

Returns new model instance

data: Any

dict(*, *include: Optional[Union[AbstractSetIntStr, MappingIntStrAny]] = None*, *exclude: Optional[Union[AbstractSetIntStr, MappingIntStrAny]] = None*, *by_alias: bool = False*, *skip_defaults: Optional[bool] = None*, *exclude_unset: bool = False*, *exclude_defaults: bool = False*, *exclude_none: bool = False*) → DictStrAny

Generate a dictionary representation of the model, optionally specifying which fields to include or exclude.

domain: Dict

features() → List

classmethod from_constraints(*constraints:* synthcity.plugins.core.constraints.Constraints) → *synthcity.plugins.core.schema.Schema*

Create a schema from a list of Constraints.

classmethod from_orm(*obj: Any*) → Model

get(*feature: str*) → *synthcity.plugins.core.distribution.Distribution*

Get the Distribution of a feature.

Parameters **feature** – str. the feature name

Returns The feature distribution

includes(*other:* synthcity.plugins.core.schema.Schema) → bool

Test if another schema is included in the local one.

json(*, *include: Optional[Union[AbstractSetIntStr, MappingIntStrAny]] = None*, *exclude: Optional[Union[AbstractSetIntStr, MappingIntStrAny]] = None*, *by_alias: bool = False*, *skip_defaults: Optional[bool] = None*, *exclude_unset: bool = False*, *exclude_defaults: bool = False*, *exclude_none: bool = False*, *encoder: Optional[Callable[[Any], Any]] = None*, *models_as_dict: bool = True*, ***dumps_kwargs: Any*) → unicode

Generate a JSON representation of the model, *include* and *exclude* arguments as per *dict()*.

encoder is an optional function to supply as *default* to json.dumps(), other arguments as per *json.dumps()*.

classmethod parse_file(*path: Union[str, pathlib.Path]*, ***, *content_type: unicode = None*, *encoding: unicode = 'utf8'*, *proto: pydantic.parse.Protocol = None*, *allow_pickle: bool = False*) → Model

classmethod parse_obj(*obj: Any*) → Model

classmethod parse_raw(*b: Union[str, bytes]*, ***, *content_type: unicode = None*, *encoding: unicode = 'utf8'*, *proto: pydantic.parse.Protocol = None*, *allow_pickle: bool = False*) → Model

protected_cols: List[str]

random_state: int

sample(*count: int*) → pandas.core.frame.DataFrame

sampling_strategy: str

classmethod schema(*by_alias: bool = True*, *ref_template: unicode = '#/definitions/{model}'*) → DictStrAny

classmethod schema_json(*, *by_alias: bool = True*, *ref_template: unicode = '#/definitions/{model}'*, ***dumps_kwargs: Any*) → unicode

classmethod update_forward_refs(***localns: Any*) → None
Try to update ForwardRefs on fields based on this Model, globalns and localns.

classmethod validate(*value: Any*) → Model

synthcity.plugins.core.serializable module

class Serializable(**args: Any*, ***kwargs: Any*)
Bases: `object`

Utility class for model persistence.

static load(*buff: bytes*) → Any

static load_dict(*representation: dict*) → Any

save() → bytes

save_dict() → dict

save_to_file(*path: pathlib.Path*) → bytes

static version() → str
API version

12.1.2 Core model implementations

synthcity.plugins.core.models.tabular_encoder module

TabularEncoder module.

class BinEncoder(**args: Any*, ***kwargs: Any*)
Bases: *synthcity.plugins.core.models.tabular_encoder.TabularEncoder*

Binary encoder (for SurvivalGAN).

Model continuous columns with a BayesianGMM and normalized to a scalar [0, 1] and a vector. Discrete columns are encoded using a scikit-learn OneHotEncoder.

activation_layout(*discrete_activation: str*, *continuous_activation: str*) → Sequence[Tuple[str, int]]
Get the layout of the activations.

Returns a list of tuple, describing each column as:

- continuous, and with length 1 + number of GMM clusters.
- discrete, and with length <N>, the length of the one-hot encoding.

cat_encoder_params: dict = {}

categorical_encoder: Union[str, type] = 'passthrough'

cont_encoder_params: dict = {'n_components': 2}

continuous_encoder: Union[str, type] = 'bayesian_gmm'

fit(*raw_data: pandas.core.frame.DataFrame*, *discrete_columns: Optional[List] = None*) → Any
Fit the TabularEncoder.

This step also counts the #columns in matrix data and span information.

get_column_info(*name: str*) → *synthcity.plugins.core.models.tabular_encoder.FeatureInfo*

inverse_transform(*data: pandas.core.frame.DataFrame*) → pandas.core.frame.DataFrame
Take matrix data and output raw data.

Output uses the same type as input to the transform function.

layout() → Sequence[*synthcity.plugins.core.models.tabular_encoder.FeatureInfo*]
Get the layout of the encoded dataset.

Returns a list of tuple, describing each column as:

- continuous, and with length 1 + number of GMM clusters.
- discrete, and with length <N>, the length of the one-hot encoding.

n_features() → int

transform(*raw_data: pandas.core.frame.DataFrame*) → pandas.core.frame.DataFrame
Take raw data and output a matrix data.

class FeatureInfo(**, name: str, feature_type: str, transform: Any = None, output_dimensions: int, transformed_features: List[str], trans_feature_types: List[str]*)
Bases: pydantic.main.BaseModel

Config
alias of pydantic.config.BaseConfig

classmethod construct(*_fields_set: Optional[SetStr] = None*, ***values: Any*) → Model
Creates a new model setting __dict__ and __fields_set__ from trusted or pre-validated data. Default values are respected, but no other validation is performed. Behaves as if *Config.extra = 'allow'* was set since it adds all passed values

copy(**, include: Optional[Union[AbstractSetIntStr, MappingIntStrAny]] = None*, *exclude: Optional[Union[AbstractSetIntStr, MappingIntStrAny]] = None*, *update: Optional[DictStrAny] = None*, *deep: bool = False*) → Model
Duplicate a model, optionally choose which fields to include, exclude and change.

Parameters

- **include** – fields to include in new model
- **exclude** – fields to exclude from new model, as with values this takes precedence over include
- **update** – values to change/add in the new model. Note: the data is not validated before creating the new model: you should trust this data
- **deep** – set to *True* to make a deep copy of the model

Returns new model instance

dict(**, include: Optional[Union[AbstractSetIntStr, MappingIntStrAny]] = None*, *exclude: Optional[Union[AbstractSetIntStr, MappingIntStrAny]] = None*, *by_alias: bool = False*, *skip_defaults: Optional[bool] = None*, *exclude_unset: bool = False*, *exclude_defaults: bool = False*, *exclude_none: bool = False*) → DictStrAny
Generate a dictionary representation of the model, optionally specifying which fields to include or exclude.

feature_type: str

classmethod from_orm(*obj: Any*) → Model

json(*, *include: Optional[Union[AbstractSetIntStr, MappingIntStrAny]] = None*, *exclude: Optional[Union[AbstractSetIntStr, MappingIntStrAny]] = None*, *by_alias: bool = False*, *skip_defaults: Optional[bool] = None*, *exclude_unset: bool = False*, *exclude_defaults: bool = False*, *exclude_none: bool = False*, *encoder: Optional[Callable[[Any], Any]] = None*, *models_as_dict: bool = True*, ***dumps_kwargs: Any*) → unicode

Generate a JSON representation of the model, *include* and *exclude* arguments as per *dict()*.

encoder is an optional function to supply as *default* to json.dumps(), other arguments as per *json.dumps()*.

name: str

output_dimensions: int

classmethod parse_file(*path: Union[str, pathlib.Path]*, ***, *content_type: unicode = None*, *encoding: unicode = 'utf8'*, *proto: pydantic.parse.Protocol = None*, *allow_pickle: bool = False*) → Model

classmethod parse_obj(*obj: Any*) → Model

classmethod parse_raw(*b: Union[str, bytes]*, ***, *content_type: unicode = None*, *encoding: unicode = 'utf8'*, *proto: pydantic.parse.Protocol = None*, *allow_pickle: bool = False*) → Model

classmethod schema(*by_alias: bool = True*, *ref_template: unicode = '#/definitions/{model}'*) → DictStrAny

classmethod schema_json(*, *by_alias: bool = True*, *ref_template: unicode = '#/definitions/{model}'*, ***dumps_kwargs: Any*) → unicode

trans_feature_types: List[str]

transform: Any

transformed_features: List[str]

classmethod update_forward_refs(***localns: Any*) → None

Try to update ForwardRefs on fields based on this Model, globalns and localns.

classmethod validate(*value: Any*) → Model

class TabularEncoder(**args: Any*, ***kwargs: Any*)

Bases: `sklearn.base.TransformerMixin`, `sklearn.base.BaseEstimator`

Tabular encoder.

Model continuous columns with a BayesianGMM and normalized to a scalar [0, 1] and a vector. Discrete columns are encoded using a scikit-learn OneHotEncoder.

activation_layout(*discrete_activation: str*, *continuous_activation: str*) → Sequence[Tuple[str, int]]

Get the layout of the activations.

Returns a list of tuple, describing each column as:

- continuous, and with length 1 + number of GMM clusters.
- discrete, and with length <N>, the length of the one-hot encoding.

cat_encoder_params: dict = {'handle_unknown': 'ignore', 'sparse_output': False}

categorical_encoder: Union[str, type] = 'onehot'

cont_encoder_params: dict = {'n_components': 10}

continuous_encoder: Union[str, type] = 'bayesian_gmm'

fit(*raw_data: pandas.core.frame.DataFrame*, *discrete_columns: Optional[List] = None*) → Any
Fit the TabularEncoder.

This step also counts the #columns in matrix data and span information.

get_column_info(*name: str*) → *synthcity.plugins.core.models.tabular_encoder.FeatureInfo*

inverse_transform(*data: pandas.core.frame.DataFrame*) → pandas.core.frame.DataFrame
Take matrix data and output raw data.

Output uses the same type as input to the transform function.

layout() → Sequence[*synthcity.plugins.core.models.tabular_encoder.FeatureInfo*]
Get the layout of the encoded dataset.

Returns a list of tuple, describing each column as:

- continuous, and with length 1 + number of GMM clusters.
- discrete, and with length <N>, the length of the one-hot encoding.

n_features() → int

transform(*raw_data: pandas.core.frame.DataFrame*) → pandas.core.frame.DataFrame
Take raw data and output a matrix data.

class TimeSeriesBinEncoder(**args: Any*, ***kwargs: Any*)
Bases: sklearn.base.TransformerMixin, sklearn.base.BaseEstimator

Time series Bin encoder.

Model continuous columns with a BayesianGMM and normalized to a scalar [0, 1] and a vector. Discrete columns are encoded using a scikit-learn OneHotEncoder.

fit(*static_data: pandas.core.frame.DataFrame*, *temporal_data: List[pandas.core.frame.DataFrame]*, *observation_times: List*, *discrete_columns: Optional[List] = None*) → *synthcity.plugins.core.models.tabular_encoder.TimeSeriesBinEncoder*
Fit the TimeSeriesBinEncoder

fit_transform(*static: pandas.core.frame.DataFrame*, *temporal: List[pandas.core.frame.DataFrame]*, *observation_times: List*) → pandas.core.frame.DataFrame

transform(*static_data: pandas.core.frame.DataFrame*, *temporal_data: List[pandas.core.frame.DataFrame]*, *observation_times: List*) → pandas.core.frame.DataFrame
Take raw data and output a matrix data.

class TimeSeriesTabularEncoder(**args: Any*, ***kwargs: Any*)
Bases: sklearn.base.TransformerMixin, sklearn.base.BaseEstimator

TimeSeries Tabular encoder.

Model continuous columns with a BayesianGMM and normalized to a scalar [0, 1] and a vector. Discrete columns are encoded using a scikit-learn OneHotEncoder.

activation_layout(*discrete_activation: str*, *continuous_activation: str*) → Tuple

activation_layout_temporal(*discrete_activation: str*, *continuous_activation: str*) → Any

fit(*static_data: pandas.core.frame.DataFrame*, *temporal_data: List[pandas.core.frame.DataFrame]*, *observation_times: List*, *discrete_columns: Optional[List] = None*) → *synthcity.plugins.core.models.tabular_encoder.TimeSeriesTabularEncoder*

fit_temporal(*temporal_data: List[pandas.core.frame.DataFrame]*, *observation_times: List*, *discrete_columns: Optional[List] = None*) → *synthcity.plugins.core.models.tabular_encoder.TimeSeriesTabularEncoder*

fit_transform(*static_data: pandas.core.frame.DataFrame*, *temporal_data: List[pandas.core.frame.DataFrame]*, *observation_times: List*) → Tuple[pandas.core.frame.DataFrame, pandas.core.frame.DataFrame, List]

fit_transform_temporal(*temporal_data: List[pandas.core.frame.DataFrame]*, *observation_times: List*) → Tuple[pandas.core.frame.DataFrame, List]

inverse_transform(*static_encoded: pandas.core.frame.DataFrame*, *temporal_encoded: List[pandas.core.frame.DataFrame]*, *observation_times: List*) → pandas.core.frame.DataFrame

inverse_transform_observation_times(*observation_times: List*) → pandas.core.frame.DataFrame

inverse_transform_static(*static_encoded: pandas.core.frame.DataFrame*) → pandas.core.frame.DataFrame

inverse_transform_temporal(*temporal_encoded: List[pandas.core.frame.DataFrame]*, *observation_times: List*) → pandas.core.frame.DataFrame

layout() → Tuple[List, List]

n_features() → Tuple

transform(*static_data: pandas.core.frame.DataFrame*, *temporal_data: List[pandas.core.frame.DataFrame]*, *observation_times: List*) → Tuple[pandas.core.frame.DataFrame, pandas.core.frame.DataFrame, List]

transform_observation_times(*observation_times: List*) → List

transform_static(*static_data: pandas.core.frame.DataFrame*) → pandas.core.frame.DataFrame

transform_temporal(*temporal_data: List[pandas.core.frame.DataFrame]*, *observation_times: List*) → Tuple[pandas.core.frame.DataFrame, List]

synthcity.plugins.core.models.mlp module

class LinearLayer(*n_units_in: int*, *n_units_out: int*, *dropout: float = 0*, *batch_norm: bool = False*, *nonlin: Optional[str] = 'relu'*, *device: Any = device(type='cpu')*)

Bases: `torch.nn.modules.module.Module`

T_destination

alias of TypeVar('T_destination', bound=`Dict[str, Any]`)

add_module(*name: str*, *module: Optional[torch.nn.modules.module.Module]*) → None

Adds a child module to the current module.

The module can be accessed as an attribute using the given name.

Parameters

- **name** (*str*) – name of the child module. The child module can be accessed from this module using the given name
- **module** (*Module*) – child module to be added to the module.

apply(*fn: Callable[[torch.nn.modules.module.Module], None]*) → torch.nn.modules.module.T

Applies `fn` recursively to every submodule (as returned by `.children()`) as well as self. Typical use includes initializing the parameters of a model (see also nn-init-doc).

Parameters **fn** (`Module` -> None) – function to be applied to each submodule

Returns self

Return type Module

Example:

```
>>> @torch.no_grad()
>>> def init_weights(m):
>>>     print(m)
>>>     if type(m) == nn.Linear:
>>>         m.weight.fill_(1.0)
>>>         print(m.weight)
>>> net = nn.Sequential(nn.Linear(2, 2), nn.Linear(2, 2))
>>> net.apply(init_weights)
Linear(in_features=2, out_features=2, bias=True)
Parameter containing:
tensor([[1., 1.],
        [1., 1.]], requires_grad=True)
Linear(in_features=2, out_features=2, bias=True)
Parameter containing:
tensor([[1., 1.],
        [1., 1.]], requires_grad=True)
Sequential(
  (0): Linear(in_features=2, out_features=2, bias=True)
  (1): Linear(in_features=2, out_features=2, bias=True)
)
```

bfloat16() → torch.nn.modules.module.T
Casts all floating point parameters and buffers to `bfloat16` datatype.

Note: This method modifies the module in-place.

Returns self

Return type Module

buffers(*recurse: bool = True*) → Iterator[torch.Tensor]
Returns an iterator over module buffers.

Parameters **recurse** (*bool*) – if True, then yields buffers of this module and all submodules. Otherwise, yields only buffers that are direct members of this module.

Yields *torch.Tensor* – module buffer

Example:

```
>>> # xdoctest: +SKIP("undefined vars")
>>> for buf in model.buffers():
>>>     print(type(buf), buf.size())
<class 'torch.Tensor'> (20L,)
<class 'torch.Tensor'> (20L, 1L, 5L, 5L)
```

children() → Iterator[torch.nn.modules.module.Module]
Returns an iterator over immediate children modules.

Yields *Module* – a child module

cpu() → torch.nn.modules.module.T
Moves all model parameters and buffers to the CPU.

Note: This method modifies the module in-place.

Returns self

Return type Module

cuda(*device: Optional[Union[int, torch.device]] = None*) → torch.nn.modules.module.T
Moves all model parameters and buffers to the GPU.

This also makes associated parameters and buffers different objects. So it should be called before constructing optimizer if the module will live on GPU while being optimized.

Note: This method modifies the module in-place.

Parameters **device** (*int, optional*) – if specified, all parameters will be copied to that device

Returns self

Return type Module

double() → torch.nn.modules.module.T
Casts all floating point parameters and buffers to `double` datatype.

Note: This method modifies the module in-place.

Returns self

Return type Module

dump_patches: bool = False

eval() → torch.nn.modules.module.T
Sets the module in evaluation mode.

This has any effect only on certain modules. See documentations of particular modules for details of their behaviors in training/evaluation mode, if they are affected, e.g. `Dropout`, `BatchNorm`, etc.

This is equivalent with `self.train(False)`.

See locally-disable-grad-doc for a comparison between *.eval()* and several similar mechanisms that may be confused with it.

Returns self

Return type Module

extra_repr() → str
Set the extra representation of the module

To print customized extra information, you should re-implement this method in your own modules. Both single-line and multi-line strings are acceptable.

float() → torch.nn.modules.module.T
Casts all floating point parameters and buffers to `float` datatype.

Note: This method modifies the module in-place.

Returns self

Return type Module

forward(*X: torch.Tensor*) → torch.Tensor
Defines the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

get_buffer(*target: str*) → torch.Tensor
Returns the buffer given by `target` if it exists, otherwise throws an error.

See the docstring for `get_submodule` for a more detailed explanation of this method's functionality as well as how to correctly specify `target`.

Parameters **target** – The fully-qualified string name of the buffer to look for. (See `get_submodule` for how to specify a fully-qualified string.)

Returns The buffer referenced by `target`

Return type torch.Tensor

Raises **AttributeError** – If the target string references an invalid path or resolves to something that is not a buffer

get_extra_state() → Any
Returns any extra state to include in the module's state_dict. Implement this and a corresponding *set_extra_state()* for your module if you need to store extra state. This function is called when building the module's *state_dict()*.

Note that extra state should be pickleable to ensure working serialization of the state_dict. We only provide provide backwards compatibility guarantees for serializing Tensors; other objects may break backwards compatibility if their serialized pickled form changes.

Returns Any extra state to store in the module's state_dict

Return type object

get_parameter(*target: str*) → torch.nn.parameter.Parameter
Returns the parameter given by `target` if it exists, otherwise throws an error.

See the docstring for `get_submodule` for a more detailed explanation of this method's functionality as well as how to correctly specify `target`.

Parameters **target** – The fully-qualified string name of the Parameter to look for. (See `get_submodule` for how to specify a fully-qualified string.)

Returns The Parameter referenced by `target`

Return type torch.nn.Parameter

Raises **AttributeError** – If the target string references an invalid path or resolves to something that is not an `nn.Parameter`

get_submodule(*target: str*) → torch.nn.modules.module.Module
Returns the submodule given by `target` if it exists, otherwise throws an error.

For example, let's say you have an `nn.Module` `A` that looks like this:

```
A(
    (net_b): Module(
        (net_c): Module(
            (conv): Conv2d(16, 33, kernel_size=(3, 3), stride=(2, 2))
        )
        (linear): Linear(in_features=100, out_features=200, bias=True)
    )
)
```

(The diagram shows an `nn.Module` `A`. `A` has a nested submodule `net_b`, which itself has two submodules `net_c` and `linear`. `net_c` then has a submodule `conv`.)

To check whether or not we have the `linear` submodule, we would call `get_submodule("net_b.linear")`. To check whether we have the `conv` submodule, we would call `get_submodule("net_b.net_c.conv")`.

The runtime of `get_submodule` is bounded by the degree of module nesting in `target`. A query against `named_modules` achieves the same result, but it is O(N) in the number of transitive modules. So, for a simple check to see if some submodule exists, `get_submodule` should always be used.

Parameters `target` – The fully-qualified string name of the submodule to look for. (See above example for how to specify a fully-qualified string.)

Returns The submodule referenced by `target`

Return type torch.nn.Module

Raises `AttributeError` – If the target string references an invalid path or resolves to something that is not an `nn.Module`

half() → torch.nn.modules.module.T
Casts all floating point parameters and buffers to `half` datatype.

Note: This method modifies the module in-place.

Returns self

Return type Module

ipu(*device: Optional[Union[int, torch.device]] = None*) → torch.nn.modules.module.T
Moves all model parameters and buffers to the IPU.

This also makes associated parameters and buffers different objects. So it should be called before constructing optimizer if the module will live on IPU while being optimized.

Note: This method modifies the module in-place.

Parameters `device` (*int, optional*) – if specified, all parameters will be copied to that device

Returns self

Return type Module

load_state_dict(*state_dict: Mapping[str, Any]*, *strict: bool = True*)
Copies parameters and buffers from *state_dict* into this module and its descendants. If `strict` is `True`, then the keys of *state_dict* must exactly match the keys returned by this module's `state_dict()` function.

Parameters

- **state_dict** (*dict*) – a dict containing parameters and persistent buffers.
- **strict** (*bool, optional*) – whether to strictly enforce that the keys in *state_dict* match the keys returned by this module's `state_dict()` function. Default: `True`

Returns

- **missing_keys** is a list of str containing the missing keys
- **unexpected_keys** is a list of str containing the unexpected keys

Return type `NamedTuple` with `missing_keys` and `unexpected_keys` fields

Note: If a parameter or buffer is registered as `None` and its corresponding key exists in *state_dict*, *load_state_dict()* will raise a `RuntimeError`.

---

modules() → Iterator[torch.nn.modules.module.Module]
Returns an iterator over all modules in the network.

Yields *Module* – a module in the network

Note: Duplicate modules are returned only once. In the following example, `l` will be returned only once.

Example:

```
>>> l = nn.Linear(2, 2)
>>> net = nn.Sequential(l, l)
>>> for idx, m in enumerate(net.modules()):
...     print(idx, '->', m)

0 -> Sequential(
  (0): Linear(in_features=2, out_features=2, bias=True)
  (1): Linear(in_features=2, out_features=2, bias=True)
)
1 -> Linear(in_features=2, out_features=2, bias=True)
```

named_buffers(*prefix: str = ''*, *recurse: bool = True*) → Iterator[Tuple[str, torch.Tensor]]
Returns an iterator over module buffers, yielding both the name of the buffer as well as the buffer itself.

Parameters

- **prefix** (*str*) – prefix to prepend to all buffer names.
- **recurse** (*bool*) – if True, then yields buffers of this module and all submodules. Otherwise, yields only buffers that are direct members of this module.

Yields *(str, torch.Tensor)* – Tuple containing the name and buffer

Example:

```
>>> # xdoctest: +SKIP("undefined vars")
>>> for name, buf in self.named_buffers():
>>>    if name in ['running_var']:
>>>        print(buf.size())
```

named_children() → Iterator[Tuple[str, torch.nn.modules.module.Module]]

Returns an iterator over immediate children modules, yielding both the name of the module as well as the module itself.

Yields *(str, Module)* – Tuple containing a name and child module

Example:

```
>>> # xdoctest: +SKIP("undefined vars")
>>> for name, module in model.named_children():
>>>     if name in ['conv4', 'conv5']:
>>>         print(module)
```

named_modules(*memo: Optional[Set[torch.nn.modules.module.Module]] = None*, *prefix: str = ''*, *remove_duplicate: bool = True*)

Returns an iterator over all modules in the network, yielding both the name of the module as well as the module itself.

Parameters

- **memo** – a memo to store the set of modules already added to the result
- **prefix** – a prefix that will be added to the name of the module
- **remove_duplicate** – whether to remove the duplicated module instances in the result or not

Yields *(str, Module)* – Tuple of name and module

Note: Duplicate modules are returned only once. In the following example, `l` will be returned only once.

Example:

```
>>> l = nn.Linear(2, 2)
>>> net = nn.Sequential(l, l)
>>> for idx, m in enumerate(net.named_modules()):
...     print(idx, '->', m)

0 -> ('', Sequential(
  (0): Linear(in_features=2, out_features=2, bias=True)
  (1): Linear(in_features=2, out_features=2, bias=True)
))
1 -> ('0', Linear(in_features=2, out_features=2, bias=True))
```

named_parameters(*prefix: str = ''*, *recurse: bool = True*) → Iterator[Tuple[str, torch.nn.parameter.Parameter]]

Returns an iterator over module parameters, yielding both the name of the parameter as well as the parameter itself.

Parameters

- **prefix** (*str*) – prefix to prepend to all parameter names.

- **recurse** (*bool*) – if True, then yields parameters of this module and all submodules. Otherwise, yields only parameters that are direct members of this module.

Yields *(str, Parameter)* – Tuple containing the name and parameter

Example:

```
>>> # xdoctest: +SKIP("undefined vars")
>>> for name, param in self.named_parameters():
>>>    if name in ['bias']:
>>>        print(param.size())
```

parameters(*recurse: bool = True*) → Iterator[torch.nn.parameter.Parameter]

Returns an iterator over module parameters.

This is typically passed to an optimizer.

Parameters **recurse** (*bool*) – if True, then yields parameters of this module and all submodules. Otherwise, yields only parameters that are direct members of this module.

Yields *Parameter* – module parameter

Example:

```
>>> # xdoctest: +SKIP("undefined vars")
>>> for param in model.parameters():
>>>     print(type(param), param.size())
<class 'torch.Tensor'> (20L,)
<class 'torch.Tensor'> (20L, 1L, 5L, 5L)
```

register_backward_hook(*hook: Callable[[torch.nn.modules.module.Module, Union[Tuple[torch.Tensor, ...], torch.Tensor], Union[Tuple[torch.Tensor, ...], torch.Tensor]], Union[None, torch.Tensor]]*) → torch.utils.hooks.RemovableHandle

Registers a backward hook on the module.

This function is deprecated in favor of `register_full_backward_hook()` and the behavior of this function will change in future versions.

Returns a handle that can be used to remove the added hook by calling `handle.remove()`

Return type `torch.utils.hooks.RemovableHandle`

register_buffer(*name: str*, *tensor: Optional[torch.Tensor]*, *persistent: bool = True*) → None

Adds a buffer to the module.

This is typically used to register a buffer that should not to be considered a model parameter. For example, BatchNorm's `running_mean` is not a parameter, but is part of the module's state. Buffers, by default, are persistent and will be saved alongside parameters. This behavior can be changed by setting `persistent` to `False`. The only difference between a persistent buffer and a non-persistent buffer is that the latter will not be a part of this module's *state_dict*.

Buffers can be accessed as attributes using given names.

Parameters

- **name** (*str*) – name of the buffer. The buffer can be accessed from this module using the given name
- **tensor** (*Tensor or None*) – buffer to be registered. If `None`, then operations that run on buffers, such as *cuda*, are ignored. If `None`, the buffer is **not** included in the module's *state_dict*.

- **persistent** (*bool*) – whether the buffer is part of this module's *state_dict*.

Example:

```
>>> # xdoctest: +SKIP("undefined vars")
>>> self.register_buffer('running_mean', torch.zeros(num_features))
```

register_forward_hook(*hook: Callable[[...], None]*) → torch.utils.hooks.RemovableHandle

Registers a forward hook on the module.

The hook will be called every time after *forward()* has computed an output. It should have the following signature:

```
hook(module, input, output) -> None or modified output
```

The input contains only the positional arguments given to the module. Keyword arguments won't be passed to the hooks and only to the `forward`. The hook can modify the output. It can modify the input inplace but it will not have effect on forward since this is called after *forward()* is called.

Returns a handle that can be used to remove the added hook by calling `handle.remove()`

Return type `torch.utils.hooks.RemovableHandle`

register_forward_pre_hook(*hook: Callable[[...], None]*) → torch.utils.hooks.RemovableHandle

Registers a forward pre-hook on the module.

The hook will be called every time before *forward()* is invoked. It should have the following signature:

```
hook(module, input) -> None or modified input
```

The input contains only the positional arguments given to the module. Keyword arguments won't be passed to the hooks and only to the `forward`. The hook can modify the input. User can either return a tuple or a single modified value in the hook. We will wrap the value into a tuple if a single value is returned(unless that value is already a tuple).

Returns a handle that can be used to remove the added hook by calling `handle.remove()`

Return type `torch.utils.hooks.RemovableHandle`

register_full_backward_hook(*hook: Callable[[torch.nn.modules.module.Module, Union[Tuple[torch.Tensor, ...], torch.Tensor], Union[Tuple[torch.Tensor, ...], torch.Tensor]], Union[None, torch.Tensor]]*) → torch.utils.hooks.RemovableHandle

Registers a backward hook on the module.

The hook will be called every time the gradients with respect to a module are computed, i.e. the hook will execute if and only if the gradients with respect to module outputs are computed. The hook should have the following signature:

```
hook(module, grad_input, grad_output) -> tuple(Tensor) or None
```

The `grad_input` and `grad_output` are tuples that contain the gradients with respect to the inputs and outputs respectively. The hook should not modify its arguments, but it can optionally return a new gradient with respect to the input that will be used in place of `grad_input` in subsequent computations. `grad_input` will only correspond to the inputs given as positional arguments and all kwarg arguments are ignored. Entries in `grad_input` and `grad_output` will be `None` for all non-Tensor arguments.

For technical reasons, when this hook is applied to a Module, its forward function will receive a view of each Tensor passed to the Module. Similarly the caller will receive a view of each Tensor returned by the Module's forward function.

> **Warning:** Modifying inputs or outputs inplace is not allowed when using backward hooks and will raise an error.

Returns a handle that can be used to remove the added hook by calling `handle.remove()`

Return type `torch.utils.hooks.RemovableHandle`

register_load_state_dict_post_hook(*hook*)

Registers a post hook to be run after module's `load_state_dict` is called.

It should have the following signature:: hook(module, incompatible_keys) -> None

The `module` argument is the current module that this hook is registered on, and the `incompatible_keys` argument is a `NamedTuple` consisting of attributes `missing_keys` and `unexpected_keys`. `missing_keys` is a `list` of `str` containing the missing keys and `unexpected_keys` is a `list` of `str` containing the unexpected keys.

The given incompatible_keys can be modified inplace if needed.

Note that the checks performed when calling *load_state_dict()* with `strict=True` are affected by modifications the hook makes to `missing_keys` or `unexpected_keys`, as expected. Additions to either set of keys will result in an error being thrown when `strict=True`, and clearning out both missing and unexpected keys will avoid an error.

Returns a handle that can be used to remove the added hook by calling `handle.remove()`

Return type `torch.utils.hooks.RemovableHandle`

register_module(*name: str*, *module: Optional[torch.nn.modules.module.Module]*) → None

Alias for *add_module()*.

register_parameter(*name: str*, *param: Optional[torch.nn.parameter.Parameter]*) → None

Adds a parameter to the module.

The parameter can be accessed as an attribute using given name.

Parameters

- **name** (*str*) – name of the parameter. The parameter can be accessed from this module using the given name
- **param** (*Parameter or None*) – parameter to be added to the module. If `None`, then operations that run on parameters, such as *cuda*, are ignored. If `None`, the parameter is **not** included in the module's *state_dict*.

requires_grad_(*requires_grad: bool = True*) → torch.nn.modules.module.T

Change if autograd should record operations on parameters in this module.

This method sets the parameters' `requires_grad` attributes in-place.

This method is helpful for freezing part of the module for finetuning or training parts of a model individually (e.g., GAN training).

See locally-disable-grad-doc for a comparison between *.requires_grad_()* and several similar mechanisms that may be confused with it.

Parameters **requires_grad** (*bool*) – whether autograd should record operations on parameters in this module. Default: `True`.

Returns self

Return type Module

set_extra_state(*state: Any*)
This function is called from *load_state_dict()* to handle any extra state found within the *state_dict*. Implement this function and a corresponding *get_extra_state()* for your module if you need to store extra state within its *state_dict*.

Parameters **state** (*dict*) – Extra state from the *state_dict*

share_memory() → torch.nn.modules.module.T
See `torch.Tensor.share_memory_()`

state_dict(**args*, *destination=None*, *prefix=''*, *keep_vars=False*)
Returns a dictionary containing references to the whole state of the module.

Both parameters and persistent buffers (e.g. running averages) are included. Keys are corresponding parameter and buffer names. Parameters and buffers set to `None` are not included.

Note: The returned object is a shallow copy. It contains references to the module's parameters and buffers.

Warning: Currently `state_dict()` also accepts positional arguments for `destination`, `prefix` and `keep_vars` in order. However, this is being deprecated and keyword arguments will be enforced in future releases.

Warning: Please avoid the use of argument `destination` as it is not designed for end-users.

Parameters

- **destination** (*dict, optional*) – If provided, the state of module will be updated into the dict and the same object is returned. Otherwise, an `OrderedDict` will be created and returned. Default: `None`.
- **prefix** (*str, optional*) – a prefix added to parameter and buffer names to compose the keys in state_dict. Default: `''`.
- **keep_vars** (*bool, optional*) – by default the `Tensor` s returned in the state dict are detached from autograd. If it's set to `True`, detaching will not be performed. Default: `False`.

Returns a dictionary containing a whole state of the module

Return type dict

Example:

```
>>> # xdoctest: +SKIP("undefined vars")
>>> module.state_dict().keys()
['bias', 'weight']
```

to(**args*, ***kwargs*)
Moves and/or casts the parameters and buffers.

This can be called as

to(*device=None*, *dtype=None*, *non_blocking=False*)

to(*dtype*, *non_blocking=False*)

to(*tensor*, *non_blocking=False*)

to(*memory_format=torch.channels_last*)

Its signature is similar to `torch.Tensor.to()`, but only accepts floating point or complex `dtype`s. In addition, this method will only cast the floating point or complex parameters and buffers to `dtype` (if given). The integral parameters and buffers will be moved `device`, if that is given, but with dtypes unchanged. When `non_blocking` is set, it tries to convert/move asynchronously with respect to the host if possible, e.g., moving CPU Tensors with pinned memory to CUDA devices.

See below for examples.

Note: This method modifies the module in-place.

Parameters

- **device** (`torch.device`) – the desired device of the parameters and buffers in this module
- **dtype** (`torch.dtype`) – the desired floating point or complex dtype of the parameters and buffers in this module
- **tensor** (*torch.Tensor*) – Tensor whose dtype and device are the desired dtype and device for all parameters and buffers in this module
- **memory_format** (`torch.memory_format`) – the desired memory format for 4D parameters and buffers in this module (keyword only argument)

Returns self

Return type Module

Examples:

```
>>> # xdoctest: +IGNORE_WANT("non-deterministic")
>>> linear = nn.Linear(2, 2)
>>> linear.weight
Parameter containing:
tensor([[ 0.1913, -0.3420],
        [-0.5113, -0.2325]])
>>> linear.to(torch.double)
Linear(in_features=2, out_features=2, bias=True)
>>> linear.weight
Parameter containing:
tensor([[ 0.1913, -0.3420],
        [-0.5113, -0.2325]], dtype=torch.float64)
>>> # xdoctest: +REQUIRES(env:TORCH_DOCTEST_CUDA1)
>>> gpu1 = torch.device("cuda:1")
>>> linear.to(gpu1, dtype=torch.half, non_blocking=True)
Linear(in_features=2, out_features=2, bias=True)
>>> linear.weight
Parameter containing:
tensor([[ 0.1914, -0.3420],
        [-0.5112, -0.2324]], dtype=torch.float16, device='cuda:1')
>>> cpu = torch.device("cpu")
>>> linear.to(cpu)
```

(continues on next page)

(continued from previous page)

```
Linear(in_features=2, out_features=2, bias=True)
>>> linear.weight
Parameter containing:
tensor([[ 0.1914, -0.3420],
        [-0.5112, -0.2324]], dtype=torch.float16)

>>> linear = nn.Linear(2, 2, bias=None).to(torch.cdouble)
>>> linear.weight
Parameter containing:
tensor([[ 0.3741+0.j,  0.2382+0.j],
        [ 0.5593+0.j, -0.4443+0.j]], dtype=torch.complex128)
>>> linear(torch.ones(3, 2, dtype=torch.cdouble))
tensor([[0.6122+0.j, 0.1150+0.j],
        [0.6122+0.j, 0.1150+0.j],
        [0.6122+0.j, 0.1150+0.j]], dtype=torch.complex128)
```

to_empty(*, *device: Union[str, torch.device]*) → torch.nn.modules.module.T
Moves the parameters and buffers to the specified device without copying storage.

Parameters **device** (`torch.device`) – The desired device of the parameters and buffers in this module.

Returns self

Return type Module

train(*mode: bool = True*) → torch.nn.modules.module.T
Sets the module in training mode.

This has any effect only on certain modules. See documentations of particular modules for details of their behaviors in training/evaluation mode, if they are affected, e.g. `Dropout`, `BatchNorm`, etc.

Parameters **mode** (*bool*) – whether to set training mode (`True`) or evaluation mode (`False`). Default: `True`.

Returns self

Return type Module

training: bool

type(*dst_type: Union[torch.dtype, str]*) → torch.nn.modules.module.T
Casts all parameters and buffers to `dst_type`.

Note: This method modifies the module in-place.

Parameters **dst_type** (*type or string*) – the desired type

Returns self

Return type Module

xpu(*device: Optional[Union[int, torch.device]] = None*) → torch.nn.modules.module.T
Moves all model parameters and buffers to the XPU.

This also makes associated parameters and buffers different objects. So it should be called before constructing optimizer if the module will live on XPU while being optimized.

Note: This method modifies the module in-place.

Parameters **device** (*int, optional*) – if specified, all parameters will be copied to that device

Returns self

Return type Module

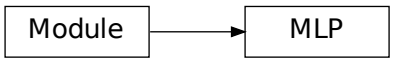
zero_grad(*set_to_none: bool = False*) → None

Sets gradients of all model parameters to zero. See similar function under `torch.optim.Optimizer` for more context.

Parameters **set_to_none** (*bool*) – instead of setting to zero, set the grads to None. See `torch.optim.Optimizer.zero_grad()` for details.

class MLP(*n_units_in: int*, *n_units_out: int*, ***, *task_type: str = 'regression'*, *n_layers_hidden: int = 1*, *n_units_hidden: int = 100*, *nonlin: str = 'relu'*, *nonlin_out: Optional[List[Tuple[str, int]]] = None*, *lr: float = 0.001*, *weight_decay: float = 0.001*, *opt_betas: tuple = (0.9, 0.999)*, *n_iter: int = 1000*, *batch_size: int = 500*, *n_iter_print: int = 100*, *random_state: int = 0*, *patience: int = 10*, *n_iter_min: int = 100*, *dropout: float = 0.1*, *clipping_value: int = 1*, *batch_norm: bool = False*, *early_stopping: bool = True*, *residual: bool = False*, *loss: Optional[Callable] = None*, *device: Any = device(type='cpu')*)

Bases: `torch.nn.modules.module.Module`



Fully connected or residual neural nets for classification and regression.

Parameters

- **task_type** (*str*) – classification or regression
- **n_units_int** (*int*) – Number of features
- **n_units_out** (*int*) – Number of outputs
- **n_layers_hidden** (*int*) – Number of hidden layers
- **n_units_hidden** (*int*) – Number of hidden units in each layer
- **nonlin** (*string, default 'elu'*) – Nonlinearity to use in NN. Can be 'elu', 'relu', 'selu', 'tanh' or 'leaky_relu'.
- **lr** (*float*) – learning rate for optimizer.
- **weight_decay** (*float*) – l2 (ridge) penalty for the weights.
- **n_iter** (*int*) – Maximum number of iterations.
- **batch_size** (*int*) – Batch size
- **n_iter_print** (*int*) – Number of iterations after which to print updates and check the validation loss.

- **random_state** (*int*) – random_state used
- **patience** (*int*) – Number of iterations to wait before early stopping after decrease in validation loss
- **n_iter_min** (*int*) – Minimum number of iterations to go through before starting early stopping
- **dropout** (*float*) – Dropout value. If 0, the dropout is not used.
- **clipping_value** (*int, default 1*) – Gradients clipping value
- **batch_norm** (*bool*) – Enable/disable batch norm
- **early_stopping** (*bool*) – Enable/disable early stopping
- **residual** (*bool*) – Add residuals.
- **loss** (*Callable*) – Optional Custom loss function. If None, the loss is CrossEntropy for classification tasks, or RMSE for regression.

T_destination
alias of TypeVar('T_destination', bound=Dict[str, Any])

add_module(*name: str*, *module: Optional[torch.nn.modules.module.Module]*) → None
Adds a child module to the current module.

The module can be accessed as an attribute using the given name.

Parameters

- **name** (*str*) – name of the child module. The child module can be accessed from this module using the given name
- **module** (*Module*) – child module to be added to the module.

apply(*fn: Callable[[torch.nn.modules.module.Module], None]*) → torch.nn.modules.module.T
Applies `fn` recursively to every submodule (as returned by `.children()`) as well as self. Typical use includes initializing the parameters of a model (see also nn-init-doc).

Parameters **fn** (Module -> None) – function to be applied to each submodule

Returns self

Return type Module

Example:

```
>>> @torch.no_grad()
>>> def init_weights(m):
>>>     print(m)
>>>     if type(m) == nn.Linear:
>>>         m.weight.fill_(1.0)
>>>         print(m.weight)
>>> net = nn.Sequential(nn.Linear(2, 2), nn.Linear(2, 2))
>>> net.apply(init_weights)
Linear(in_features=2, out_features=2, bias=True)
Parameter containing:
tensor([[1., 1.],
        [1., 1.]], requires_grad=True)
Linear(in_features=2, out_features=2, bias=True)
Parameter containing:
tensor([[1., 1.],
```

(continues on next page)

(continued from previous page)

```
        [1., 1.]], requires_grad=True)
Sequential(
  (0): Linear(in_features=2, out_features=2, bias=True)
  (1): Linear(in_features=2, out_features=2, bias=True)
)
```

bfloat16() → torch.nn.modules.module.T
Casts all floating point parameters and buffers to `bfloat16` datatype.

Note: This method modifies the module in-place.

Returns self

Return type Module

buffers(*recurse: bool = True*) → Iterator[torch.Tensor]
Returns an iterator over module buffers.

Parameters **recurse** (*bool*) – if True, then yields buffers of this module and all submodules. Otherwise, yields only buffers that are direct members of this module.

Yields *torch.Tensor* – module buffer

Example:

```
>>> # xdoctest: +SKIP("undefined vars")
>>> for buf in model.buffers():
>>>     print(type(buf), buf.size())
<class 'torch.Tensor'> (20L,)
<class 'torch.Tensor'> (20L, 1L, 5L, 5L)
```

children() → Iterator[torch.nn.modules.module.Module]
Returns an iterator over immediate children modules.

Yields *Module* – a child module

cpu() → torch.nn.modules.module.T
Moves all model parameters and buffers to the CPU.

Note: This method modifies the module in-place.

Returns self

Return type Module

cuda(*device: Optional[Union[int, torch.device]] = None*) → torch.nn.modules.module.T
Moves all model parameters and buffers to the GPU.

This also makes associated parameters and buffers different objects. So it should be called before constructing optimizer if the module will live on GPU while being optimized.

Note: This method modifies the module in-place.

Parameters **device** (*int, optional*) – if specified, all parameters will be copied to that device

Returns self

Return type Module

double() → torch.nn.modules.module.T
Casts all floating point parameters and buffers to double datatype.

Note: This method modifies the module in-place.

Returns self

Return type Module

dump_patches: bool = False

eval() → torch.nn.modules.module.T
Sets the module in evaluation mode.

This has any effect only on certain modules. See documentations of particular modules for details of their behaviors in training/evaluation mode, if they are affected, e.g. Dropout, BatchNorm, etc.

This is equivalent with self.train(False).

See locally-disable-grad-doc for a comparison between *.eval()* and several similar mechanisms that may be confused with it.

Returns self

Return type Module

extra_repr() → str
Set the extra representation of the module

To print customized extra information, you should re-implement this method in your own modules. Both single-line and multi-line strings are acceptable.

fit(*X: numpy.ndarray*, *y: numpy.ndarray*) → *synthcity.plugins.core.models.mlp.MLP*

float() → torch.nn.modules.module.T
Casts all floating point parameters and buffers to float datatype.

Note: This method modifies the module in-place.

Returns self

Return type Module

forward(*X: torch.Tensor*) → torch.Tensor
Defines the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

get_buffer(*target: str*) → torch.Tensor
Returns the buffer given by `target` if it exists, otherwise throws an error.

See the docstring for `get_submodule` for a more detailed explanation of this method's functionality as well as how to correctly specify `target`.

Parameters **target** – The fully-qualified string name of the buffer to look for. (See `get_submodule` for how to specify a fully-qualified string.)

Returns The buffer referenced by `target`

Return type torch.Tensor

Raises **AttributeError** – If the target string references an invalid path or resolves to something that is not a buffer

get_extra_state() → Any
Returns any extra state to include in the module's state_dict. Implement this and a corresponding *set_extra_state()* for your module if you need to store extra state. This function is called when building the module's *state_dict()*.

Note that extra state should be pickleable to ensure working serialization of the state_dict. We only provide provide backwards compatibility guarantees for serializing Tensors; other objects may break backwards compatibility if their serialized pickled form changes.

Returns Any extra state to store in the module's state_dict

Return type object

get_parameter(*target: str*) → torch.nn.parameter.Parameter
Returns the parameter given by `target` if it exists, otherwise throws an error.

See the docstring for `get_submodule` for a more detailed explanation of this method's functionality as well as how to correctly specify `target`.

Parameters **target** – The fully-qualified string name of the Parameter to look for. (See `get_submodule` for how to specify a fully-qualified string.)

Returns The Parameter referenced by `target`

Return type torch.nn.Parameter

Raises **AttributeError** – If the target string references an invalid path or resolves to something that is not an `nn.Parameter`

get_submodule(*target: str*) → torch.nn.modules.module.Module
Returns the submodule given by `target` if it exists, otherwise throws an error.

For example, let's say you have an `nn.Module` `A` that looks like this:

```
A(
    (net_b): Module(
        (net_c): Module(
            (conv): Conv2d(16, 33, kernel_size=(3, 3), stride=(2, 2))
        )
        (linear): Linear(in_features=100, out_features=200, bias=True)
```

(continues on next page)

(continued from previous page)

```
    )
)
```

(The diagram shows an `nn.Module` `A`. `A` has a nested submodule `net_b`, which itself has two submodules `net_c` and `linear`. `net_c` then has a submodule `conv`.)

To check whether or not we have the `linear` submodule, we would call `get_submodule("net_b.linear")`. To check whether we have the `conv` submodule, we would call `get_submodule("net_b.net_c.conv")`.

The runtime of `get_submodule` is bounded by the degree of module nesting in `target`. A query against `named_modules` achieves the same result, but it is O(N) in the number of transitive modules. So, for a simple check to see if some submodule exists, `get_submodule` should always be used.

Parameters `target` – The fully-qualified string name of the submodule to look for. (See above example for how to specify a fully-qualified string.)

Returns The submodule referenced by `target`

Return type torch.nn.Module

Raises `AttributeError` – If the target string references an invalid path or resolves to something that is not an `nn.Module`

`half()` → torch.nn.modules.module.T

Casts all floating point parameters and buffers to `half` datatype.

Note: This method modifies the module in-place.

Returns self

Return type Module

`ipu(`*device: Optional[Union[int, torch.device]] = None*`)` → torch.nn.modules.module.T

Moves all model parameters and buffers to the IPU.

This also makes associated parameters and buffers different objects. So it should be called before constructing optimizer if the module will live on IPU while being optimized.

Note: This method modifies the module in-place.

Parameters `device` (*int, optional*) – if specified, all parameters will be copied to that device

Returns self

Return type Module

`load_state_dict(`*state_dict: Mapping[str, Any]*, *strict: bool = True*`)`

Copies parameters and buffers from *state_dict* into this module and its descendants. If `strict` is `True`, then the keys of *state_dict* must exactly match the keys returned by this module's `state_dict()` function.

Parameters

- `state_dict` (*dict*) – a dict containing parameters and persistent buffers.

- **strict** (*bool, optional*) – whether to strictly enforce that the keys in *state_dict* match the keys returned by this module's `state_dict()` function. Default: `True`

Returns

- **missing_keys** is a list of str containing the missing keys
- **unexpected_keys** is a list of str containing the unexpected keys

Return type `NamedTuple` with `missing_keys` and `unexpected_keys` fields

Note: If a parameter or buffer is registered as `None` and its corresponding key exists in *state_dict*, *load_state_dict()* will raise a `RuntimeError`.

---

modules() → Iterator[torch.nn.modules.module.Module]

Returns an iterator over all modules in the network.

Yields *Module* – a module in the network

Note: Duplicate modules are returned only once. In the following example, `l` will be returned only once.

Example:

```
>>> l = nn.Linear(2, 2)
>>> net = nn.Sequential(l, l)
>>> for idx, m in enumerate(net.modules()):
...     print(idx, '->', m)

0 -> Sequential(
  (0): Linear(in_features=2, out_features=2, bias=True)
  (1): Linear(in_features=2, out_features=2, bias=True)
)
1 -> Linear(in_features=2, out_features=2, bias=True)
```

named_buffers(*prefix: str = ''*, *recurse: bool = True*) → Iterator[Tuple[str, torch.Tensor]]

Returns an iterator over module buffers, yielding both the name of the buffer as well as the buffer itself.

Parameters

- **prefix** (*str*) – prefix to prepend to all buffer names.
- **recurse** (*bool*) – if True, then yields buffers of this module and all submodules. Otherwise, yields only buffers that are direct members of this module.

Yields *(str, torch.Tensor)* – Tuple containing the name and buffer

Example:

```
>>> # xdoctest: +SKIP("undefined vars")
>>> for name, buf in self.named_buffers():
>>>    if name in ['running_var']:
>>>        print(buf.size())
```

named_children() → Iterator[Tuple[str, torch.nn.modules.module.Module]]

Returns an iterator over immediate children modules, yielding both the name of the module as well as the module itself.

Yields *(str, Module)* – Tuple containing a name and child module

Example:

```
>>> # xdoctest: +SKIP("undefined vars")
>>> for name, module in model.named_children():
>>>     if name in ['conv4', 'conv5']:
>>>         print(module)
```

named_modules(*memo: Optional[Set[torch.nn.modules.module.Module]] = None*, *prefix: str = ''*, *remove_duplicate: bool = True*)

Returns an iterator over all modules in the network, yielding both the name of the module as well as the module itself.

Parameters

- **memo** – a memo to store the set of modules already added to the result
- **prefix** – a prefix that will be added to the name of the module
- **remove_duplicate** – whether to remove the duplicated module instances in the result or not

Yields *(str, Module)* – Tuple of name and module

Note: Duplicate modules are returned only once. In the following example, `l` will be returned only once.

Example:

```
>>> l = nn.Linear(2, 2)
>>> net = nn.Sequential(l, l)
>>> for idx, m in enumerate(net.named_modules()):
...     print(idx, '->', m)

0 -> ('', Sequential(
  (0): Linear(in_features=2, out_features=2, bias=True)
  (1): Linear(in_features=2, out_features=2, bias=True)
))
1 -> ('0', Linear(in_features=2, out_features=2, bias=True))
```

named_parameters(*prefix: str = ''*, *recurse: bool = True*) → Iterator[Tuple[str, torch.nn.parameter.Parameter]]

Returns an iterator over module parameters, yielding both the name of the parameter as well as the parameter itself.

Parameters

- **prefix** (*str*) – prefix to prepend to all parameter names.
- **recurse** (*bool*) – if True, then yields parameters of this module and all submodules. Otherwise, yields only parameters that are direct members of this module.

Yields *(str, Parameter)* – Tuple containing the name and parameter

Example:

```
>>> # xdoctest: +SKIP("undefined vars")
>>> for name, param in self.named_parameters():
>>>    if name in ['bias']:
>>>        print(param.size())
```

parameters(*recurse: bool = True*) → Iterator[torch.nn.parameter.Parameter]
Returns an iterator over module parameters.

This is typically passed to an optimizer.

Parameters **recurse** (*bool*) – if True, then yields parameters of this module and all submodules. Otherwise, yields only parameters that are direct members of this module.

Yields *Parameter* – module parameter

Example:

```
>>> # xdoctest: +SKIP("undefined vars")
>>> for param in model.parameters():
>>>     print(type(param), param.size())
<class 'torch.Tensor'> (20L,)
<class 'torch.Tensor'> (20L, 1L, 5L, 5L)
```

predict(*X: numpy.ndarray*) → numpy.ndarray

predict_proba(*X: numpy.ndarray*) → numpy.ndarray

register_backward_hook(*hook: Callable[[torch.nn.modules.module.Module, Union[Tuple[torch.Tensor, ...], torch.Tensor], Union[Tuple[torch.Tensor, ...], torch.Tensor]], Union[None, torch.Tensor]]*) → torch.utils.hooks.RemovableHandle
Registers a backward hook on the module.

This function is deprecated in favor of `register_full_backward_hook()` and the behavior of this function will change in future versions.

Returns a handle that can be used to remove the added hook by calling `handle.remove()`

Return type `torch.utils.hooks.RemovableHandle`

register_buffer(*name: str*, *tensor: Optional[torch.Tensor]*, *persistent: bool = True*) → None
Adds a buffer to the module.

This is typically used to register a buffer that should not to be considered a model parameter. For example, BatchNorm's `running_mean` is not a parameter, but is part of the module's state. Buffers, by default, are persistent and will be saved alongside parameters. This behavior can be changed by setting `persistent` to `False`. The only difference between a persistent buffer and a non-persistent buffer is that the latter will not be a part of this module's *state_dict*.

Buffers can be accessed as attributes using given names.

Parameters

- **name** (*str*) – name of the buffer. The buffer can be accessed from this module using the given name
- **tensor** (*Tensor or None*) – buffer to be registered. If `None`, then operations that run on buffers, such as *cuda*, are ignored. If `None`, the buffer is **not** included in the module's *state_dict*.
- **persistent** (*bool*) – whether the buffer is part of this module's *state_dict*.

Example:

```
>>> # xdoctest: +SKIP("undefined vars")
>>> self.register_buffer('running_mean', torch.zeros(num_features))
```

register_forward_hook(*hook: Callable[[...], None]*) → torch.utils.hooks.RemovableHandle
Registers a forward hook on the module.

The hook will be called every time after *forward()* has computed an output. It should have the following signature:

```
hook(module, input, output) -> None or modified output
```

The input contains only the positional arguments given to the module. Keyword arguments won't be passed to the hooks and only to the `forward`. The hook can modify the output. It can modify the input inplace but it will not have effect on forward since this is called after *forward()* is called.

Returns a handle that can be used to remove the added hook by calling `handle.remove()`

Return type `torch.utils.hooks.RemovableHandle`

register_forward_pre_hook(*hook: Callable[[...], None]*) → torch.utils.hooks.RemovableHandle

Registers a forward pre-hook on the module.

The hook will be called every time before *forward()* is invoked. It should have the following signature:

```
hook(module, input) -> None or modified input
```

The input contains only the positional arguments given to the module. Keyword arguments won't be passed to the hooks and only to the `forward`. The hook can modify the input. User can either return a tuple or a single modified value in the hook. We will wrap the value into a tuple if a single value is returned(unless that value is already a tuple).

Returns a handle that can be used to remove the added hook by calling `handle.remove()`

Return type `torch.utils.hooks.RemovableHandle`

register_full_backward_hook(*hook: Callable[[torch.nn.modules.module.Module, Union[Tuple[torch.Tensor, ...], torch.Tensor], Union[Tuple[torch.Tensor, ...], torch.Tensor]], Union[None, torch.Tensor]]*) → torch.utils.hooks.RemovableHandle

Registers a backward hook on the module.

The hook will be called every time the gradients with respect to a module are computed, i.e. the hook will execute if and only if the gradients with respect to module outputs are computed. The hook should have the following signature:

```
hook(module, grad_input, grad_output) -> tuple(Tensor) or None
```

The `grad_input` and `grad_output` are tuples that contain the gradients with respect to the inputs and outputs respectively. The hook should not modify its arguments, but it can optionally return a new gradient with respect to the input that will be used in place of `grad_input` in subsequent computations. `grad_input` will only correspond to the inputs given as positional arguments and all kwarg arguments are ignored. Entries in `grad_input` and `grad_output` will be `None` for all non-Tensor arguments.

For technical reasons, when this hook is applied to a Module, its forward function will receive a view of each Tensor passed to the Module. Similarly the caller will receive a view of each Tensor returned by the Module's forward function.

Warning: Modifying inputs or outputs inplace is not allowed when using backward hooks and will raise an error.

Returns a handle that can be used to remove the added hook by calling `handle.remove()`

Return type `torch.utils.hooks.RemovableHandle`

register_load_state_dict_post_hook(*hook*)
Registers a post hook to be run after module's `load_state_dict` is called.

It should have the following signature:: hook(module, incompatible_keys) -> None

The `module` argument is the current module that this hook is registered on, and the `incompatible_keys` argument is a `NamedTuple` consisting of attributes `missing_keys` and `unexpected_keys`. `missing_keys` is a `list` of `str` containing the missing keys and `unexpected_keys` is a `list` of `str` containing the unexpected keys.

The given incompatible_keys can be modified inplace if needed.

Note that the checks performed when calling *load_state_dict()* with `strict=True` are affected by modifications the hook makes to `missing_keys` or `unexpected_keys`, as expected. Additions to either set of keys will result in an error being thrown when `strict=True`, and clearning out both missing and unexpected keys will avoid an error.

Returns a handle that can be used to remove the added hook by calling `handle.remove()`

Return type `torch.utils.hooks.RemovableHandle`

register_module(*name: str*, *module: Optional[torch.nn.modules.module.Module]*) → None
Alias for *add_module()*.

register_parameter(*name: str*, *param: Optional[torch.nn.parameter.Parameter]*) → None
Adds a parameter to the module.

The parameter can be accessed as an attribute using given name.

Parameters

- **name** (*str*) – name of the parameter. The parameter can be accessed from this module using the given name
- **param** (*Parameter or None*) – parameter to be added to the module. If `None`, then operations that run on parameters, such as *cuda*, are ignored. If `None`, the parameter is **not** included in the module's *state_dict*.

requires_grad_(*requires_grad: bool = True*) → torch.nn.modules.module.T
Change if autograd should record operations on parameters in this module.

This method sets the parameters' `requires_grad` attributes in-place.

This method is helpful for freezing part of the module for finetuning or training parts of a model individually (e.g., GAN training).

See locally-disable-grad-doc for a comparison between *.requires_grad_()* and several similar mechanisms that may be confused with it.

Parameters **requires_grad** (*bool*) – whether autograd should record operations on parameters in this module. Default: `True`.

Returns self

Return type Module

score(*X: numpy.ndarray*, *y: numpy.ndarray*) → *float*

set_extra_state(*state: Any*)
This function is called from *load_state_dict()* to handle any extra state found within the *state_dict*. Implement this function and a corresponding *get_extra_state()* for your module if you need to store extra state within its *state_dict*.

Parameters **state** (*dict*) – Extra state from the *state_dict*

share_memory() → torch.nn.modules.module.T
See `torch.Tensor.share_memory_()`

state_dict(**args*, *destination=None*, *prefix=''*, *keep_vars=False*)
Returns a dictionary containing references to the whole state of the module.

Both parameters and persistent buffers (e.g. running averages) are included. Keys are corresponding parameter and buffer names. Parameters and buffers set to `None` are not included.

Note: The returned object is a shallow copy. It contains references to the module's parameters and buffers.

Warning: Currently `state_dict()` also accepts positional arguments for `destination`, `prefix` and `keep_vars` in order. However, this is being deprecated and keyword arguments will be enforced in future releases.

Warning: Please avoid the use of argument `destination` as it is not designed for end-users.

Parameters

- **destination** (*dict, optional*) – If provided, the state of module will be updated into the dict and the same object is returned. Otherwise, an `OrderedDict` will be created and returned. Default: `None`.
- **prefix** (*str, optional*) – a prefix added to parameter and buffer names to compose the keys in state_dict. Default: `''`.
- **keep_vars** (*bool, optional*) – by default the `Tensor` s returned in the state dict are detached from autograd. If it's set to `True`, detaching will not be performed. Default: `False`.

Returns a dictionary containing a whole state of the module

Return type dict

Example:

```
>>> # xdoctest: +SKIP("undefined vars")
>>> module.state_dict().keys()
['bias', 'weight']
```

to(**args*, ***kwargs*)
Moves and/or casts the parameters and buffers.

This can be called as

to(*device=None*, *dtype=None*, *non_blocking=False*)

to(*dtype*, *non_blocking=False*)

to(*tensor*, *non_blocking=False*)

to(*memory_format=torch.channels_last*)

Its signature is similar to `torch.Tensor.to()`, but only accepts floating point or complex `dtype`s. In addition, this method will only cast the floating point or complex parameters and buffers to `dtype` (if given). The integral parameters and buffers will be moved `device`, if that is given, but with dtypes unchanged.

When non_blocking is set, it tries to convert/move asynchronously with respect to the host if possible, e.g., moving CPU Tensors with pinned memory to CUDA devices.

See below for examples.

Note: This method modifies the module in-place.

Parameters

- **device** (torch.device) – the desired device of the parameters and buffers in this module
- **dtype** (torch.dtype) – the desired floating point or complex dtype of the parameters and buffers in this module
- **tensor** (*torch.Tensor*) – Tensor whose dtype and device are the desired dtype and device for all parameters and buffers in this module
- **memory_format** (torch.memory_format) – the desired memory format for 4D parameters and buffers in this module (keyword only argument)

Returns self

Return type Module

Examples:

```
>>> # xdoctest: +IGNORE_WANT("non-deterministic")
>>> linear = nn.Linear(2, 2)
>>> linear.weight
Parameter containing:
tensor([[ 0.1913, -0.3420],
        [-0.5113, -0.2325]])
>>> linear.to(torch.double)
Linear(in_features=2, out_features=2, bias=True)
>>> linear.weight
Parameter containing:
tensor([[ 0.1913, -0.3420],
        [-0.5113, -0.2325]], dtype=torch.float64)
>>> # xdoctest: +REQUIRES(env:TORCH_DOCTEST_CUDA1)
>>> gpu1 = torch.device("cuda:1")
>>> linear.to(gpu1, dtype=torch.half, non_blocking=True)
Linear(in_features=2, out_features=2, bias=True)
>>> linear.weight
Parameter containing:
tensor([[ 0.1914, -0.3420],
        [-0.5112, -0.2324]], dtype=torch.float16, device='cuda:1')
>>> cpu = torch.device("cpu")
>>> linear.to(cpu)
Linear(in_features=2, out_features=2, bias=True)
>>> linear.weight
Parameter containing:
tensor([[ 0.1914, -0.3420],
        [-0.5112, -0.2324]], dtype=torch.float16)
```

(continues on next page)

(continued from previous page)

```
>>> linear = nn.Linear(2, 2, bias=None).to(torch.cdouble)
>>> linear.weight
Parameter containing:
tensor([[ 0.3741+0.j,  0.2382+0.j],
        [ 0.5593+0.j, -0.4443+0.j]], dtype=torch.complex128)
>>> linear(torch.ones(3, 2, dtype=torch.cdouble))
tensor([[0.6122+0.j, 0.1150+0.j],
        [0.6122+0.j, 0.1150+0.j],
        [0.6122+0.j, 0.1150+0.j]], dtype=torch.complex128)
```

to_empty(*, *device: Union[str, torch.device]*) → torch.nn.modules.module.T
Moves the parameters and buffers to the specified device without copying storage.

Parameters **device** (torch.device) – The desired device of the parameters and buffers in this module.

Returns self

Return type Module

train(*mode: bool = True*) → torch.nn.modules.module.T
Sets the module in training mode.

This has any effect only on certain modules. See documentations of particular modules for details of their behaviors in training/evaluation mode, if they are affected, e.g. Dropout, BatchNorm, etc.

Parameters **mode** (*bool*) – whether to set training mode (True) or evaluation mode (False). Default: True.

Returns self

Return type Module

training: bool

type(*dst_type: Union[torch.dtype, str]*) → torch.nn.modules.module.T
Casts all parameters and buffers to dst_type.

Note: This method modifies the module in-place.

Parameters **dst_type** (*type or string*) – the desired type

Returns self

Return type Module

xpu(*device: Optional[Union[int, torch.device]] = None*) → torch.nn.modules.module.T
Moves all model parameters and buffers to the XPU.

This also makes associated parameters and buffers different objects. So it should be called before constructing optimizer if the module will live on XPU while being optimized.

Note: This method modifies the module in-place.

Parameters **device** (*int, optional*) – if specified, all parameters will be copied to that device

Returns self

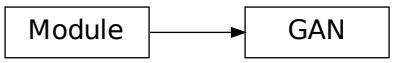
Return type Module

zero_grad(*set_to_none: bool = False*) → None
Sets gradients of all model parameters to zero. See similar function under `torch.optim.Optimizer` for more context.

Parameters **set_to_none** (*bool*) – instead of setting to zero, set the grads to None. See `torch.optim.Optimizer.zero_grad()` for details.

synthcity.plugins.core.models.gan module

class GAN(*n_features: int*, *n_units_latent: int*, *n_units_conditional: int = 0*, *generator_n_layers_hidden: int = 2*, *generator_n_units_hidden: int = 250*, *generator_nonlin: str = 'leaky_relu'*, *generator_nonlin_out: Optional[List[Tuple[str, int]]] = None*, *generator_n_iter: int = 500*, *generator_batch_norm: bool = False*, *generator_dropout: float = 0*, *generator_lr: float = 0.0002*, *generator_weight_decay: float = 0.001*, *generator_residual: bool = True*, *generator_opt_betas: tuple = (0.9, 0.999)*, *generator_extra_penalties: list = []*, *generator_extra_penalty_cbks: List[Callable] = []*, *discriminator_n_layers_hidden: int = 3*, *discriminator_n_units_hidden: int = 300*, *discriminator_nonlin: str = 'leaky_relu'*, *discriminator_n_iter: int = 1*, *discriminator_batch_norm: bool = False*, *discriminator_dropout: float = 0.1*, *discriminator_lr: float = 0.0002*, *discriminator_weight_decay: float = 0.001*, *discriminator_opt_betas: tuple = (0.9, 0.999)*, *batch_size: int = 64*, *random_state: int = 0*, *clipping_value: int = 0*, *lambda_gradient_penalty: float = 10*, *lambda_identifiability_penalty: float = 0.1*, *dataloader_sampler: Optional[torch.utils.data.sampler.Sampler] = None*, *device: Any = device(type='cpu')*, *n_iter_min: int = 100*, *n_iter_print: int = 10*, *patience: int = 20*, *patience_metric: Optional[synthcity.metrics.weighted_metrics.WeightedMetrics] = None*, *dp_enabled: bool = False*, *dp_delta: Optional[float] = None*, *dp_epsilon: float = 3*, *dp_max_grad_norm: float = 2*, *dp_secure_mode: bool = False*)
Bases: `torch.nn.modules.module.Module`



Basic GAN implementation.

Parameters

- **n_features** – int Number of features
- **n_units_latent** – int Number of latent units
- **n_units_conditional** – int Number of conditional units
- **generator_n_layers_hidden** – int Number of hidden layers in the generator
- **generator_n_units_hidden** – int Number of hidden units in each layer of the Generator
- **generator_nonlin** – string, default 'elu' Nonlinearity to use in the generator. Can be 'elu', 'relu', 'selu' or 'leaky_relu'.

- **generator_n_iter** – int Maximum number of iterations in the Generator.
- **generator_batch_norm** – bool Enable/disable batch norm for the generator
- **generator_dropout** – float Dropout value. If 0, the dropout is not used.
- **generator_residual** – bool Use residuals for the generator
- **generator_nonlin_out** – Optional[List[Tuple[str, int]]] List of activations. Useful with the TabularEncoder
- **generator_lr** – float = 2e-4 Generator learning rate, used by the Adam optimizer
- **generator_weight_decay** – float = 1e-3 Generator weight decay, used by the Adam optimizer
- **generator_opt_betas** – tuple = (0.9, 0.999) Generator initial decay rates, used by the Adam Optimizer
- **generator_extra_penalties** – list Additional penalties for the generator. Values: "identifiability_penalty"
- **generator_extra_penalty_cbks** – List[Callable] Additional loss callabacks for the generator. Used by the TabularGAN for the conditional loss
- **discriminator_n_layers_hidden** – int Number of hidden layers in the discriminator
- **discriminator_n_units_hidden** – int Number of hidden units in each layer of the discriminator
- **discriminator_nonlin** – string, default 'relu' Nonlinearity to use in the discriminator. Can be 'elu', 'relu', 'selu' or 'leaky_relu'.
- **discriminator_n_iter** – int Maximum number of iterations in the discriminator.
- **discriminator_batch_norm** – bool Enable/disable batch norm for the discriminator
- **discriminator_dropout** – float Dropout value for the discriminator. If 0, the dropout is not used.
- **discriminator_lr** – float Discriminator learning rate, used by the Adam optimizer
- **discriminator_weight_decay** – float Discriminator weight decay, used by the Adam optimizer
- **discriminator_opt_betas** – tuple Initial weight decays for the Adam optimizer
- **batch_size** – int Batch size
- **random_state** – int random_state used
- **clipping_value** – int, default 0 Gradients clipping value. Zero disables the feature
- **lambda_gradient_penalty** – float = 10 Weight for the gradient penalty
- **lambda_identifiability_penalty** – float = 0.1 Weight for the identifiability penalty, if enabled
- **dataloader_sampler** – Optional[sampler.Sampler] Optional sampler for the dataloader, useful for conditional sampling
- **device** – Any = DEVICE CUDA/CPU
- **stopping** (*# early*) –
- **n_iter_print** – int Number of iterations after which to print updates and check the validation loss.

- **n_iter_min** – int Minimum number of iterations to go through before starting early stopping
- **patience** – int Max number of iterations without any improvement before early stopping is trigged.
- **patience_metric** – Optional[WeightedMetrics] If not None, the metric is used for evaluation the criterion for early stopping.
- **settings** (*# privacy*) –
- **dp_enabled** – bool Train the discriminator with Differential Privacy guarantees
- **dp_delta** – Optional[float] Optional DP delta: the probability of information accidentally being leaked. Usually 1 / len(dataset)
- **dp_epsilon** – float = 3 DP epsilon: privacy budget, which is a measure of the amount of privacy that is preserved by a given algorithm. Epsilon is a number that represents the maximum amount of information that an adversary can learn about an individual from the output of a differentially private algorithm. The smaller the value of epsilon, the more private the algorithm is. For example, an algorithm with an epsilon of 0.1 preserves more privacy than an algorithm with an epsilon of 1.0.
- **dp_max_grad_norm** – float max grad norm used for gradient clipping
- **dp_secure_mode** – bool = False, if True uses noise generation approach robust to floating point arithmetic attacks.

T_destination
alias of TypeVar('T_destination', bound=`Dict[str, Any]`)

add_module(*name: str*, *module: Optional[torch.nn.modules.module.Module]*) → None
Adds a child module to the current module.

The module can be accessed as an attribute using the given name.

Parameters

- **name** (*str*) – name of the child module. The child module can be accessed from this module using the given name
- **module** (*Module*) – child module to be added to the module.

apply(*fn: Callable[[torch.nn.modules.module.Module], None]*) → torch.nn.modules.module.T
Applies `fn` recursively to every submodule (as returned by `.children()`) as well as self. Typical use includes initializing the parameters of a model (see also nn-init-doc).

Parameters **fn** (`Module -> None`) – function to be applied to each submodule

Returns self

Return type Module

Example:

```
>>> @torch.no_grad()
>>> def init_weights(m):
>>>     print(m)
>>>     if type(m) == nn.Linear:
>>>         m.weight.fill_(1.0)
>>>         print(m.weight)
>>> net = nn.Sequential(nn.Linear(2, 2), nn.Linear(2, 2))
>>> net.apply(init_weights)
```

(continues on next page)

(continued from previous page)

```
Linear(in_features=2, out_features=2, bias=True)
Parameter containing:
tensor([[1., 1.],
        [1., 1.]], requires_grad=True)
Linear(in_features=2, out_features=2, bias=True)
Parameter containing:
tensor([[1., 1.],
        [1., 1.]], requires_grad=True)
Sequential(
  (0): Linear(in_features=2, out_features=2, bias=True)
  (1): Linear(in_features=2, out_features=2, bias=True)
)
```

bfloat16() → torch.nn.modules.module.T
Casts all floating point parameters and buffers to `bfloat16` datatype.

Note: This method modifies the module in-place.

Returns self

Return type Module

buffers(*recurse: bool = True*) → Iterator[torch.Tensor]
Returns an iterator over module buffers.

Parameters **recurse** (*bool*) – if True, then yields buffers of this module and all submodules. Otherwise, yields only buffers that are direct members of this module.

Yields *torch.Tensor* – module buffer

Example:

```
>>> # xdoctest: +SKIP("undefined vars")
>>> for buf in model.buffers():
>>>     print(type(buf), buf.size())
<class 'torch.Tensor'> (20L,)
<class 'torch.Tensor'> (20L, 1L, 5L, 5L)
```

children() → Iterator[torch.nn.modules.module.Module]
Returns an iterator over immediate children modules.

Yields *Module* – a child module

cpu() → torch.nn.modules.module.T
Moves all model parameters and buffers to the CPU.

Note: This method modifies the module in-place.

Returns self

Return type Module

cuda(*device: Optional[Union[int, torch.device]] = None*) → torch.nn.modules.module.T
Moves all model parameters and buffers to the GPU.

This also makes associated parameters and buffers different objects. So it should be called before constructing optimizer if the module will live on GPU while being optimized.

Note: This method modifies the module in-place.

Parameters **device** (*int, optional*) – if specified, all parameters will be copied to that device

Returns self

Return type Module

dataloader(*X: torch.Tensor*, *cond: Optional[torch.Tensor] = None*) →
torch.utils.data.dataloader.DataLoader

double() → torch.nn.modules.module.T
Casts all floating point parameters and buffers to `double` datatype.

Note: This method modifies the module in-place.

Returns self

Return type Module

dump_patches: bool = False

eval() → torch.nn.modules.module.T
Sets the module in evaluation mode.

This has any effect only on certain modules. See documentations of particular modules for details of their behaviors in training/evaluation mode, if they are affected, e.g. `Dropout`, `BatchNorm`, etc.

This is equivalent with `self.train(False)`.

See locally-disable-grad-doc for a comparison between *.eval()* and several similar mechanisms that may be confused with it.

Returns self

Return type Module

extra_repr() → str
Set the extra representation of the module

To print customized extra information, you should re-implement this method in your own modules. Both single-line and multi-line strings are acceptable.

fit(*X: numpy.ndarray*, *cond: Optional[numpy.ndarray] = None*, *fake_labels_generator: Optional[Callable] = None*, *true_labels_generator: Optional[Callable] = None*) → *synthcity.plugins.core.models.gan.GAN*

float() → torch.nn.modules.module.T
Casts all floating point parameters and buffers to `float` datatype.

Note: This method modifies the module in-place.

Returns self

Return type Module

forward(*count: int*, *cond: Optional[torch.Tensor] = None*) → torch.Tensor
Defines the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

---

generate(*count: int*, *cond: Optional[numpy.ndarray] = None*) → numpy.ndarray

get_buffer(*target: str*) → torch.Tensor
Returns the buffer given by `target` if it exists, otherwise throws an error.

See the docstring for `get_submodule` for a more detailed explanation of this method's functionality as well as how to correctly specify `target`.

Parameters **target** – The fully-qualified string name of the buffer to look for. (See `get_submodule` for how to specify a fully-qualified string.)

Returns The buffer referenced by `target`

Return type torch.Tensor

Raises **AttributeError** – If the target string references an invalid path or resolves to something that is not a buffer

get_extra_state() → Any
Returns any extra state to include in the module's state_dict. Implement this and a corresponding *set_extra_state()* for your module if you need to store extra state. This function is called when building the module's *state_dict()*.

Note that extra state should be pickleable to ensure working serialization of the state_dict. We only provide provide backwards compatibility guarantees for serializing Tensors; other objects may break backwards compatibility if their serialized pickled form changes.

Returns Any extra state to store in the module's state_dict

Return type object

get_parameter(*target: str*) → torch.nn.parameter.Parameter
Returns the parameter given by `target` if it exists, otherwise throws an error.

See the docstring for `get_submodule` for a more detailed explanation of this method's functionality as well as how to correctly specify `target`.

Parameters **target** – The fully-qualified string name of the Parameter to look for. (See `get_submodule` for how to specify a fully-qualified string.)

Returns The Parameter referenced by `target`

Return type torch.nn.Parameter

Raises **AttributeError** – If the target string references an invalid path or resolves to something that is not an `nn.Parameter`

get_submodule(*target: str*) → torch.nn.modules.module.Module
Returns the submodule given by `target` if it exists, otherwise throws an error.

For example, let's say you have an nn.Module A that looks like this:

```
A(
    (net_b): Module(
        (net_c): Module(
            (conv): Conv2d(16, 33, kernel_size=(3, 3), stride=(2, 2))
        )
        (linear): Linear(in_features=100, out_features=200, bias=True)
    )
)
```

(The diagram shows an nn.Module A. A has a nested submodule net_b, which itself has two submodules net_c and linear. net_c then has a submodule conv.)

To check whether or not we have the linear submodule, we would call get_submodule("net_b.linear"). To check whether we have the conv submodule, we would call get_submodule("net_b.net_c.conv").

The runtime of get_submodule is bounded by the degree of module nesting in target. A query against named_modules achieves the same result, but it is O(N) in the number of transitive modules. So, for a simple check to see if some submodule exists, get_submodule should always be used.

Parameters **target** – The fully-qualified string name of the submodule to look for. (See above example for how to specify a fully-qualified string.)

Returns The submodule referenced by target

Return type torch.nn.Module

Raises **AttributeError** – If the target string references an invalid path or resolves to something that is not an nn.Module

half() → torch.nn.modules.module.T

Casts all floating point parameters and buffers to half datatype.

Note: This method modifies the module in-place.

Returns self

Return type Module

ipu(*device: Optional[Union[int, torch.device]] = None*) → torch.nn.modules.module.T

Moves all model parameters and buffers to the IPU.

This also makes associated parameters and buffers different objects. So it should be called before constructing optimizer if the module will live on IPU while being optimized.

Note: This method modifies the module in-place.

Parameters **device** (*int, optional*) – if specified, all parameters will be copied to that device

Returns self

Return type Module

load_state_dict(*state_dict: Mapping[str, Any]*, *strict: bool = True*)

Copies parameters and buffers from *state_dict* into this module and its descendants. If `strict` is `True`, then the keys of *state_dict* must exactly match the keys returned by this module's `state_dict()` function.

Parameters

- **state_dict** (*dict*) – a dict containing parameters and persistent buffers.
- **strict** (*bool, optional*) – whether to strictly enforce that the keys in *state_dict* match the keys returned by this module's `state_dict()` function. Default: `True`

Returns

- **missing_keys** is a list of str containing the missing keys
- **unexpected_keys** is a list of str containing the unexpected keys

Return type `NamedTuple` with `missing_keys` and `unexpected_keys` fields

Note: If a parameter or buffer is registered as `None` and its corresponding key exists in *state_dict*, *load_state_dict()* will raise a `RuntimeError`.

---

modules() → Iterator[torch.nn.modules.module.Module]

Returns an iterator over all modules in the network.

Yields *Module* – a module in the network

Note: Duplicate modules are returned only once. In the following example, `l` will be returned only once.

Example:

```
>>> l = nn.Linear(2, 2)
>>> net = nn.Sequential(l, l)
>>> for idx, m in enumerate(net.modules()):
...     print(idx, '->', m)

0 -> Sequential(
  (0): Linear(in_features=2, out_features=2, bias=True)
  (1): Linear(in_features=2, out_features=2, bias=True)
)
1 -> Linear(in_features=2, out_features=2, bias=True)
```

named_buffers(*prefix: str = ''*, *recurse: bool = True*) → Iterator[Tuple[str, torch.Tensor]]

Returns an iterator over module buffers, yielding both the name of the buffer as well as the buffer itself.

Parameters

- **prefix** (*str*) – prefix to prepend to all buffer names.
- **recurse** (*bool*) – if True, then yields buffers of this module and all submodules. Otherwise, yields only buffers that are direct members of this module.

Yields *(str, torch.Tensor)* – Tuple containing the name and buffer

Example:

```
>>> # xdoctest: +SKIP("undefined vars")
>>> for name, buf in self.named_buffers():
>>>    if name in ['running_var']:
>>>        print(buf.size())
```

named_children() → Iterator[Tuple[str, torch.nn.modules.module.Module]]

Returns an iterator over immediate children modules, yielding both the name of the module as well as the module itself.

Yields *(str, Module)* – Tuple containing a name and child module

Example:

```
>>> # xdoctest: +SKIP("undefined vars")
>>> for name, module in model.named_children():
>>>     if name in ['conv4', 'conv5']:
>>>         print(module)
```

named_modules(*memo: Optional[Set[torch.nn.modules.module.Module]] = None*, *prefix: str = ''*, *remove_duplicate: bool = True*)

Returns an iterator over all modules in the network, yielding both the name of the module as well as the module itself.

Parameters

- **memo** – a memo to store the set of modules already added to the result
- **prefix** – a prefix that will be added to the name of the module
- **remove_duplicate** – whether to remove the duplicated module instances in the result or not

Yields *(str, Module)* – Tuple of name and module

Note: Duplicate modules are returned only once. In the following example, `l` will be returned only once.

Example:

```
>>> l = nn.Linear(2, 2)
>>> net = nn.Sequential(l, l)
>>> for idx, m in enumerate(net.named_modules()):
...     print(idx, '->', m)

0 -> ('', Sequential(
  (0): Linear(in_features=2, out_features=2, bias=True)
  (1): Linear(in_features=2, out_features=2, bias=True)
))
1 -> ('0', Linear(in_features=2, out_features=2, bias=True))
```

named_parameters(*prefix: str = ''*, *recurse: bool = True*) → Iterator[Tuple[str, torch.nn.parameter.Parameter]]

Returns an iterator over module parameters, yielding both the name of the parameter as well as the parameter itself.

Parameters

- **prefix** (*str*) – prefix to prepend to all parameter names.

- **recurse** (*bool*) – if True, then yields parameters of this module and all submodules. Otherwise, yields only parameters that are direct members of this module.

Yields *(str, Parameter)* – Tuple containing the name and parameter

Example:

```
>>> # xdoctest: +SKIP("undefined vars")
>>> for name, param in self.named_parameters():
>>>    if name in ['bias']:
>>>        print(param.size())
```

parameters(*recurse: bool = True*) → Iterator[torch.nn.parameter.Parameter]

Returns an iterator over module parameters.

This is typically passed to an optimizer.

Parameters **recurse** (*bool*) – if True, then yields parameters of this module and all submodules. Otherwise, yields only parameters that are direct members of this module.

Yields *Parameter* – module parameter

Example:

```
>>> # xdoctest: +SKIP("undefined vars")
>>> for param in model.parameters():
>>>     print(type(param), param.size())
<class 'torch.Tensor'> (20L,)
<class 'torch.Tensor'> (20L, 1L, 5L, 5L)
```

register_backward_hook(*hook: Callable[[torch.nn.modules.module.Module, Union[Tuple[torch.Tensor, ...], torch.Tensor], Union[Tuple[torch.Tensor, ...], torch.Tensor]], Union[None, torch.Tensor]]*) → torch.utils.hooks.RemovableHandle

Registers a backward hook on the module.

This function is deprecated in favor of `register_full_backward_hook()` and the behavior of this function will change in future versions.

Returns a handle that can be used to remove the added hook by calling `handle.remove()`

Return type `torch.utils.hooks.RemovableHandle`

register_buffer(*name: str*, *tensor: Optional[torch.Tensor]*, *persistent: bool = True*) → None

Adds a buffer to the module.

This is typically used to register a buffer that should not to be considered a model parameter. For example, BatchNorm's `running_mean` is not a parameter, but is part of the module's state. Buffers, by default, are persistent and will be saved alongside parameters. This behavior can be changed by setting `persistent` to `False`. The only difference between a persistent buffer and a non-persistent buffer is that the latter will not be a part of this module's *state_dict*.

Buffers can be accessed as attributes using given names.

Parameters

- **name** (*str*) – name of the buffer. The buffer can be accessed from this module using the given name
- **tensor** (*Tensor or None*) – buffer to be registered. If `None`, then operations that run on buffers, such as *cuda*, are ignored. If `None`, the buffer is **not** included in the module's *state_dict*.

- **persistent** (*bool*) – whether the buffer is part of this module's *state_dict*.

Example:

```
>>> # xdoctest: +SKIP("undefined vars")
>>> self.register_buffer('running_mean', torch.zeros(num_features))
```

register_forward_hook(*hook: Callable[[...], None]*) → torch.utils.hooks.RemovableHandle

Registers a forward hook on the module.

The hook will be called every time after *forward()* has computed an output. It should have the following signature:

```
hook(module, input, output) -> None or modified output
```

The input contains only the positional arguments given to the module. Keyword arguments won't be passed to the hooks and only to the `forward`. The hook can modify the output. It can modify the input inplace but it will not have effect on forward since this is called after *forward()* is called.

Returns a handle that can be used to remove the added hook by calling `handle.remove()`

Return type `torch.utils.hooks.RemovableHandle`

register_forward_pre_hook(*hook: Callable[[...], None]*) → torch.utils.hooks.RemovableHandle

Registers a forward pre-hook on the module.

The hook will be called every time before *forward()* is invoked. It should have the following signature:

```
hook(module, input) -> None or modified input
```

The input contains only the positional arguments given to the module. Keyword arguments won't be passed to the hooks and only to the `forward`. The hook can modify the input. User can either return a tuple or a single modified value in the hook. We will wrap the value into a tuple if a single value is returned(unless that value is already a tuple).

Returns a handle that can be used to remove the added hook by calling `handle.remove()`

Return type `torch.utils.hooks.RemovableHandle`

register_full_backward_hook(*hook: Callable[[torch.nn.modules.module.Module, Union[Tuple[torch.Tensor, ...], torch.Tensor], Union[Tuple[torch.Tensor, ...], torch.Tensor]], Union[None, torch.Tensor]]*) → torch.utils.hooks.RemovableHandle

Registers a backward hook on the module.

The hook will be called every time the gradients with respect to a module are computed, i.e. the hook will execute if and only if the gradients with respect to module outputs are computed. The hook should have the following signature:

```
hook(module, grad_input, grad_output) -> tuple(Tensor) or None
```

The `grad_input` and `grad_output` are tuples that contain the gradients with respect to the inputs and outputs respectively. The hook should not modify its arguments, but it can optionally return a new gradient with respect to the input that will be used in place of `grad_input` in subsequent computations. `grad_input` will only correspond to the inputs given as positional arguments and all kwarg arguments are ignored. Entries in `grad_input` and `grad_output` will be `None` for all non-Tensor arguments.

For technical reasons, when this hook is applied to a Module, its forward function will receive a view of each Tensor passed to the Module. Similarly the caller will receive a view of each Tensor returned by the Module's forward function.

> **Warning:** Modifying inputs or outputs inplace is not allowed when using backward hooks and will raise an error.

Returns a handle that can be used to remove the added hook by calling `handle.remove()`

Return type `torch.utils.hooks.RemovableHandle`

register_load_state_dict_post_hook(*hook*)

Registers a post hook to be run after module's `load_state_dict` is called.

It should have the following signature:: hook(module, incompatible_keys) -> None

The `module` argument is the current module that this hook is registered on, and the `incompatible_keys` argument is a `NamedTuple` consisting of attributes `missing_keys` and `unexpected_keys`. `missing_keys` is a `list` of `str` containing the missing keys and `unexpected_keys` is a `list` of `str` containing the unexpected keys.

The given incompatible_keys can be modified inplace if needed.

Note that the checks performed when calling *load_state_dict()* with `strict=True` are affected by modifications the hook makes to `missing_keys` or `unexpected_keys`, as expected. Additions to either set of keys will result in an error being thrown when `strict=True`, and clearning out both missing and unexpected keys will avoid an error.

Returns a handle that can be used to remove the added hook by calling `handle.remove()`

Return type `torch.utils.hooks.RemovableHandle`

register_module(*name: str*, *module: Optional[torch.nn.modules.module.Module]*) → None

Alias for *add_module()*.

register_parameter(*name: str*, *param: Optional[torch.nn.parameter.Parameter]*) → None

Adds a parameter to the module.

The parameter can be accessed as an attribute using given name.

Parameters

- **name** (*str*) – name of the parameter. The parameter can be accessed from this module using the given name
- **param** (*Parameter or None*) – parameter to be added to the module. If `None`, then operations that run on parameters, such as *cuda*, are ignored. If `None`, the parameter is **not** included in the module's *state_dict*.

requires_grad_(*requires_grad: bool = True*) → torch.nn.modules.module.T

Change if autograd should record operations on parameters in this module.

This method sets the parameters' `requires_grad` attributes in-place.

This method is helpful for freezing part of the module for finetuning or training parts of a model individually (e.g., GAN training).

See locally-disable-grad-doc for a comparison between *.requires_grad_()* and several similar mechanisms that may be confused with it.

Parameters **requires_grad** (*bool*) – whether autograd should record operations on parameters in this module. Default: `True`.

Returns self

Return type Module

set_extra_state(*state: Any*)
This function is called from *load_state_dict()* to handle any extra state found within the *state_dict*. Implement this function and a corresponding *get_extra_state()* for your module if you need to store extra state within its *state_dict*.

Parameters **state** (*dict*) – Extra state from the *state_dict*

share_memory() → torch.nn.modules.module.T
See `torch.Tensor.share_memory_()`

state_dict(**args*, *destination=None*, *prefix=''*, *keep_vars=False*)
Returns a dictionary containing references to the whole state of the module.

Both parameters and persistent buffers (e.g. running averages) are included. Keys are corresponding parameter and buffer names. Parameters and buffers set to `None` are not included.

Note: The returned object is a shallow copy. It contains references to the module's parameters and buffers.

Warning: Currently `state_dict()` also accepts positional arguments for `destination`, `prefix` and `keep_vars` in order. However, this is being deprecated and keyword arguments will be enforced in future releases.

Warning: Please avoid the use of argument `destination` as it is not designed for end-users.

Parameters

- **destination** (*dict, optional*) – If provided, the state of module will be updated into the dict and the same object is returned. Otherwise, an `OrderedDict` will be created and returned. Default: `None`.
- **prefix** (*str, optional*) – a prefix added to parameter and buffer names to compose the keys in state_dict. Default: `''`.
- **keep_vars** (*bool, optional*) – by default the `Tensor` s returned in the state dict are detached from autograd. If it's set to `True`, detaching will not be performed. Default: `False`.

Returns a dictionary containing a whole state of the module

Return type dict

Example:

```
>>> # xdoctest: +SKIP("undefined vars")
>>> module.state_dict().keys()
['bias', 'weight']
```

to(**args*, ***kwargs*)
Moves and/or casts the parameters and buffers.

This can be called as

to(*device=None*, *dtype=None*, *non_blocking=False*)

to(*dtype*, *non_blocking=False*)

to(*tensor*, *non_blocking=False*)

to(*memory_format=torch.channels_last*)

Its signature is similar to `torch.Tensor.to()`, but only accepts floating point or complex `dtype`s. In addition, this method will only cast the floating point or complex parameters and buffers to `dtype` (if given). The integral parameters and buffers will be moved `device`, if that is given, but with dtypes unchanged. When `non_blocking` is set, it tries to convert/move asynchronously with respect to the host if possible, e.g., moving CPU Tensors with pinned memory to CUDA devices.

See below for examples.

Note: This method modifies the module in-place.

Parameters

- **device** (`torch.device`) – the desired device of the parameters and buffers in this module
- **dtype** (`torch.dtype`) – the desired floating point or complex dtype of the parameters and buffers in this module
- **tensor** (*torch.Tensor*) – Tensor whose dtype and device are the desired dtype and device for all parameters and buffers in this module
- **memory_format** (`torch.memory_format`) – the desired memory format for 4D parameters and buffers in this module (keyword only argument)

Returns self

Return type Module

Examples:

```
>>> # xdoctest: +IGNORE_WANT("non-deterministic")
>>> linear = nn.Linear(2, 2)
>>> linear.weight
Parameter containing:
tensor([[ 0.1913, -0.3420],
        [-0.5113, -0.2325]])
>>> linear.to(torch.double)
Linear(in_features=2, out_features=2, bias=True)
>>> linear.weight
Parameter containing:
tensor([[ 0.1913, -0.3420],
        [-0.5113, -0.2325]], dtype=torch.float64)
>>> # xdoctest: +REQUIRES(env:TORCH_DOCTEST_CUDA1)
>>> gpu1 = torch.device("cuda:1")
>>> linear.to(gpu1, dtype=torch.half, non_blocking=True)
Linear(in_features=2, out_features=2, bias=True)
>>> linear.weight
Parameter containing:
tensor([[ 0.1914, -0.3420],
        [-0.5112, -0.2324]], dtype=torch.float16, device='cuda:1')
>>> cpu = torch.device("cpu")
>>> linear.to(cpu)
```

(continues on next page)

(continued from previous page)

```
Linear(in_features=2, out_features=2, bias=True)
>>> linear.weight
Parameter containing:
tensor([[ 0.1914, -0.3420],
        [-0.5112, -0.2324]], dtype=torch.float16)

>>> linear = nn.Linear(2, 2, bias=None).to(torch.cdouble)
>>> linear.weight
Parameter containing:
tensor([[ 0.3741+0.j,  0.2382+0.j],
        [ 0.5593+0.j, -0.4443+0.j]], dtype=torch.complex128)
>>> linear(torch.ones(3, 2, dtype=torch.cdouble))
tensor([[0.6122+0.j, 0.1150+0.j],
        [0.6122+0.j, 0.1150+0.j],
        [0.6122+0.j, 0.1150+0.j]], dtype=torch.complex128)
```

to_empty(*, *device: Union[str, torch.device]*) → torch.nn.modules.module.T
Moves the parameters and buffers to the specified device without copying storage.

Parameters **device** (`torch.device`) – The desired device of the parameters and buffers in this module.

Returns self

Return type Module

train(*mode: bool = True*) → torch.nn.modules.module.T
Sets the module in training mode.

This has any effect only on certain modules. See documentations of particular modules for details of their behaviors in training/evaluation mode, if they are affected, e.g. `Dropout`, `BatchNorm`, etc.

Parameters **mode** (*bool*) – whether to set training mode (`True`) or evaluation mode (`False`). Default: `True`.

Returns self

Return type Module

training: bool

type(*dst_type: Union[torch.dtype, str]*) → torch.nn.modules.module.T
Casts all parameters and buffers to `dst_type`.

Note: This method modifies the module in-place.

Parameters **dst_type** (*type or string*) – the desired type

Returns self

Return type Module

xpu(*device: Optional[Union[int, torch.device]] = None*) → torch.nn.modules.module.T
Moves all model parameters and buffers to the XPU.

This also makes associated parameters and buffers different objects. So it should be called before constructing optimizer if the module will live on XPU while being optimized.

Note: This method modifies the module in-place.

Parameters **device** (*int, optional*) – if specified, all parameters will be copied to that device

Returns self

Return type Module

zero_grad(*set_to_none: bool = False*) → None

Sets gradients of all model parameters to zero. See similar function under `torch.optim.Optimizer` for more context.

Parameters **set_to_none** (*bool*) – instead of setting to zero, set the grads to None. See `torch.optim.Optimizer.zero_grad()` for details.

synthcity.plugins.core.models.flows module

class NormalizingFlows(*n_iter: int = 1000*, *n_layers_hidden: int = 5*, *n_units_hidden: int = 10*, *batch_size: int = 100*, *num_transform_blocks: int = 2*, *dropout: float = 0.25*, *batch_norm: bool = False*, *num_bins: int = 8*, *tail_bound: float = 3*, *lr: float = 0.001*, *apply_unconditional_transform: bool = True*, *base_distribution: str = 'standard_normal'*, *linear_transform_type: str = 'permutation'*, *base_transform_type: str = 'rq-autoregressive'*, *device: Any = device(type='cpu')*, *n_iter_min: int = 100*, *n_iter_print: int = 10*, *patience: int = 20*, *patience_metric: Optional[synthcity.metrics.weighted_metrics.WeightedMetrics] = None*)

Bases: `torch.nn.modules.module.Module`

Normalizing Flows are generative models which produce tractable distributions where both sampling and density evaluation can be efficient and exact.

Parameters

- **n_iter** – int Number of flow steps
- **n_layers_hidden** – int Number of transformation layers
- **n_units_hidden** – int Number of hidden units for each layer
- **batch_size** – int Size of batch used for training
- **num_transform_blocks** – int Number of blocks to use in coupling/autoregressive nets.
- **dropout** – float Dropout probability for coupling/autoregressive nets.
- **batch_norm** – bool Whether to use batch norm in coupling/autoregressive nets.
- **num_bins** – int Number of bins to use for piecewise transforms.
- **tail_bound** – float Box is on [-bound, bound]^2
- **lr** – float Learning rate for optimizer.
- **apply_unconditional_transform** – bool Whether to unconditionally transform ‘identity’ features in the coupling layer.
- **base_distribution** – str Possible values: “standard_normal”
- **linear_transform_type** – str Type of linear transform to use. Possible values:
  - lu : A linear transform where we parameterize the LU decomposition of the weights.

  - permutation: Permutes using a random, but fixed, permutation.
  - svd: A linear module using the SVD decomposition for the weight matrix.
- **base_transform_type** – str Type of transform to use between linear layers. Possible values:
  - **affine-coupling** [An affine coupling layer that scales and shifts part of the variables.] Ref: L. Dinh et al., "Density estimation using Real NVP".
  - **quadratic-coupling :** Ref: Müller et al., "Neural Importance Sampling".
  - **rq-coupling** [Rational Quadratic Coupling] Ref: Durkan et al, "Neural Spline Flows".
  - **affine-autoregressive :Affine Autoregressive Transform** Ref: Durkan et al, "Neural Spline Flows".
  - **quadratic-autoregressive** [Quadratic Autoregressive Transform] Ref: Durkan et al, "Neural Spline Flows".
  - **rq-autoregressive** [Rational Quadratic Autoregressive Transform] Ref: Durkan et al, "Neural Spline Flows".
- **stopping** (*# early*) –
- **n_iter_print** – int Number of iterations after which to print updates and check the validation loss.
- **n_iter_min** – int Minimum number of iterations to go through before starting early stopping
- **patience** – int Max number of iterations without any improvement before early stopping is trigged.
- **patience_metric** – Optional[WeightedMetrics] If not None, the metric is used for evaluation the criterion for early stopping.

T_destination
alias of TypeVar('T_destination', bound=Dict[str, Any])

add_module(*name: str*, *module: Optional[torch.nn.modules.module.Module]*) → None
Adds a child module to the current module.

The module can be accessed as an attribute using the given name.

Parameters

- **name** (*str*) – name of the child module. The child module can be accessed from this module using the given name
- **module** (*Module*) – child module to be added to the module.

apply(*fn: Callable[[torch.nn.modules.module.Module], None]*) → torch.nn.modules.module.T
Applies `fn` recursively to every submodule (as returned by `.children()`) as well as self. Typical use includes initializing the parameters of a model (see also nn-init-doc).

Parameters **fn** (Module -> None) – function to be applied to each submodule

Returns self

Return type Module

Example:

```
>>> @torch.no_grad()
>>> def init_weights(m):
>>>     print(m)
>>>     if type(m) == nn.Linear:
>>>         m.weight.fill_(1.0)
>>>         print(m.weight)
>>> net = nn.Sequential(nn.Linear(2, 2), nn.Linear(2, 2))
>>> net.apply(init_weights)
Linear(in_features=2, out_features=2, bias=True)
Parameter containing:
tensor([[1., 1.],
        [1., 1.]], requires_grad=True)
Linear(in_features=2, out_features=2, bias=True)
Parameter containing:
tensor([[1., 1.],
        [1., 1.]], requires_grad=True)
Sequential(
  (0): Linear(in_features=2, out_features=2, bias=True)
  (1): Linear(in_features=2, out_features=2, bias=True)
)
```

bfloat16() → torch.nn.modules.module.T
Casts all floating point parameters and buffers to `bfloat16` datatype.

Note: This method modifies the module in-place.

Returns self

Return type Module

buffers(*recurse: bool = True*) → Iterator[torch.Tensor]
Returns an iterator over module buffers.

Parameters **recurse** (*bool*) – if True, then yields buffers of this module and all submodules. Otherwise, yields only buffers that are direct members of this module.

Yields *torch.Tensor* – module buffer

Example:

```
>>> # xdoctest: +SKIP("undefined vars")
>>> for buf in model.buffers():
>>>     print(type(buf), buf.size())
<class 'torch.Tensor'> (20L,)
<class 'torch.Tensor'> (20L, 1L, 5L, 5L)
```

children() → Iterator[torch.nn.modules.module.Module]
Returns an iterator over immediate children modules.

Yields *Module* – a child module

cpu() → torch.nn.modules.module.T
Moves all model parameters and buffers to the CPU.

Note: This method modifies the module in-place.

Returns self

Return type Module

cuda(*device: Optional[Union[int, torch.device]] = None*) → torch.nn.modules.module.T
Moves all model parameters and buffers to the GPU.

This also makes associated parameters and buffers different objects. So it should be called before constructing optimizer if the module will live on GPU while being optimized.

Note: This method modifies the module in-place.

Parameters **device** (*int, optional*) – if specified, all parameters will be copied to that device

Returns self

Return type Module

dataloader(*X: torch.Tensor*) → torch.utils.data.dataloader.DataLoader

double() → torch.nn.modules.module.T
Casts all floating point parameters and buffers to `double` datatype.

Note: This method modifies the module in-place.

Returns self

Return type Module

dump_patches: bool = False

eval() → torch.nn.modules.module.T
Sets the module in evaluation mode.

This has any effect only on certain modules. See documentations of particular modules for details of their behaviors in training/evaluation mode, if they are affected, e.g. `Dropout`, `BatchNorm`, etc.

This is equivalent with `self.train(False)`.

See locally-disable-grad-doc for a comparison between *.eval()* and several similar mechanisms that may be confused with it.

Returns self

Return type Module

extra_repr() → str
Set the extra representation of the module

To print customized extra information, you should re-implement this method in your own modules. Both single-line and multi-line strings are acceptable.

fit(*X: pandas.core.frame.DataFrame*) → Any

float() → torch.nn.modules.module.T
Casts all floating point parameters and buffers to `float` datatype.

Note: This method modifies the module in-place.

Returns self

Return type Module

forward(*count: int*) → torch.Tensor
Defines the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

---

generate(*count: int*) → numpy.ndarray

get_buffer(*target: str*) → torch.Tensor
Returns the buffer given by `target` if it exists, otherwise throws an error.

See the docstring for `get_submodule` for a more detailed explanation of this method's functionality as well as how to correctly specify `target`.

Parameters **target** – The fully-qualified string name of the buffer to look for. (See `get_submodule` for how to specify a fully-qualified string.)

Returns The buffer referenced by `target`

Return type torch.Tensor

Raises **AttributeError** – If the target string references an invalid path or resolves to something that is not a buffer

get_extra_state() → Any
Returns any extra state to include in the module's state_dict. Implement this and a corresponding *set_extra_state()* for your module if you need to store extra state. This function is called when building the module's *state_dict()*.

Note that extra state should be pickleable to ensure working serialization of the state_dict. We only provide provide backwards compatibility guarantees for serializing Tensors; other objects may break backwards compatibility if their serialized pickled form changes.

Returns Any extra state to store in the module's state_dict

Return type object

get_parameter(*target: str*) → torch.nn.parameter.Parameter
Returns the parameter given by `target` if it exists, otherwise throws an error.

See the docstring for `get_submodule` for a more detailed explanation of this method's functionality as well as how to correctly specify `target`.

Parameters **target** – The fully-qualified string name of the Parameter to look for. (See `get_submodule` for how to specify a fully-qualified string.)

Returns The Parameter referenced by `target`

Return type torch.nn.Parameter

Raises `AttributeError` – If the target string references an invalid path or resolves to something that is not an `nn.Parameter`

`get_submodule`(*target: str*) → torch.nn.modules.module.Module

Returns the submodule given by `target` if it exists, otherwise throws an error.

For example, let's say you have an `nn.Module` `A` that looks like this:

```
A(
    (net_b): Module(
        (net_c): Module(
            (conv): Conv2d(16, 33, kernel_size=(3, 3), stride=(2, 2))
        )
        (linear): Linear(in_features=100, out_features=200, bias=True)
    )
)
```

(The diagram shows an `nn.Module` `A`. `A` has a nested submodule `net_b`, which itself has two submodules `net_c` and `linear`. `net_c` then has a submodule `conv`.)

To check whether or not we have the `linear` submodule, we would call `get_submodule("net_b.linear")`. To check whether we have the `conv` submodule, we would call `get_submodule("net_b.net_c.conv")`.

The runtime of `get_submodule` is bounded by the degree of module nesting in `target`. A query against `named_modules` achieves the same result, but it is O(N) in the number of transitive modules. So, for a simple check to see if some submodule exists, `get_submodule` should always be used.

Parameters `target` – The fully-qualified string name of the submodule to look for. (See above example for how to specify a fully-qualified string.)

Returns The submodule referenced by `target`

Return type torch.nn.Module

Raises `AttributeError` – If the target string references an invalid path or resolves to something that is not an `nn.Module`

`half`() → torch.nn.modules.module.T

Casts all floating point parameters and buffers to `half` datatype.

Note: This method modifies the module in-place.

Returns self

Return type Module

`ipu`(*device: Optional[Union[int, torch.device]] = None*) → torch.nn.modules.module.T

Moves all model parameters and buffers to the IPU.

This also makes associated parameters and buffers different objects. So it should be called before constructing optimizer if the module will live on IPU while being optimized.

Note: This method modifies the module in-place.

Parameters **device** (*int, optional*) – if specified, all parameters will be copied to that device

Returns self

Return type Module

load_state_dict(*state_dict: Mapping[str, Any]*, *strict: bool = True*)

Copies parameters and buffers from *state_dict* into this module and its descendants. If `strict` is `True`, then the keys of *state_dict* must exactly match the keys returned by this module's `state_dict()` function.

Parameters

- **state_dict** (*dict*) – a dict containing parameters and persistent buffers.
- **strict** (*bool, optional*) – whether to strictly enforce that the keys in *state_dict* match the keys returned by this module's `state_dict()` function. Default: `True`

Returns

- **missing_keys** is a list of str containing the missing keys
- **unexpected_keys** is a list of str containing the unexpected keys

Return type `NamedTuple` with `missing_keys` and `unexpected_keys` fields

Note: If a parameter or buffer is registered as `None` and its corresponding key exists in *state_dict*, *load_state_dict()* will raise a `RuntimeError`.

---

modules() → Iterator[torch.nn.modules.module.Module]

Returns an iterator over all modules in the network.

Yields *Module* – a module in the network

Note: Duplicate modules are returned only once. In the following example, `l` will be returned only once.

Example:

```
>>> l = nn.Linear(2, 2)
>>> net = nn.Sequential(l, l)
>>> for idx, m in enumerate(net.modules()):
...     print(idx, '->', m)

0 -> Sequential(
  (0): Linear(in_features=2, out_features=2, bias=True)
  (1): Linear(in_features=2, out_features=2, bias=True)
)
1 -> Linear(in_features=2, out_features=2, bias=True)
```

named_buffers(*prefix: str = ''*, *recurse: bool = True*) → Iterator[Tuple[str, torch.Tensor]]

Returns an iterator over module buffers, yielding both the name of the buffer as well as the buffer itself.

Parameters

- **prefix** (*str*) – prefix to prepend to all buffer names.
- **recurse** (*bool*) – if True, then yields buffers of this module and all submodules. Otherwise, yields only buffers that are direct members of this module.

Yields *(str, torch.Tensor)* – Tuple containing the name and buffer

Example:

```
>>> # xdoctest: +SKIP("undefined vars")
>>> for name, buf in self.named_buffers():
>>>    if name in ['running_var']:
>>>        print(buf.size())
```

named_children() → Iterator[Tuple[str, torch.nn.modules.module.Module]]

Returns an iterator over immediate children modules, yielding both the name of the module as well as the module itself.

Yields *(str, Module)* – Tuple containing a name and child module

Example:

```
>>> # xdoctest: +SKIP("undefined vars")
>>> for name, module in model.named_children():
>>>     if name in ['conv4', 'conv5']:
>>>         print(module)
```

named_modules(*memo: Optional[Set[torch.nn.modules.module.Module]] = None*, *prefix: str = ''*, *remove_duplicate: bool = True*)

Returns an iterator over all modules in the network, yielding both the name of the module as well as the module itself.

Parameters

- **memo** – a memo to store the set of modules already added to the result
- **prefix** – a prefix that will be added to the name of the module
- **remove_duplicate** – whether to remove the duplicated module instances in the result or not

Yields *(str, Module)* – Tuple of name and module

Note: Duplicate modules are returned only once. In the following example, `l` will be returned only once.

Example:

```
>>> l = nn.Linear(2, 2)
>>> net = nn.Sequential(l, l)
>>> for idx, m in enumerate(net.named_modules()):
...     print(idx, '->', m)

0 -> ('', Sequential(
  (0): Linear(in_features=2, out_features=2, bias=True)
  (1): Linear(in_features=2, out_features=2, bias=True)
))
1 -> ('0', Linear(in_features=2, out_features=2, bias=True))
```

named_parameters(*prefix: str = ''*, *recurse: bool = True*) → Iterator[Tuple[str, torch.nn.parameter.Parameter]]

Returns an iterator over module parameters, yielding both the name of the parameter as well as the parameter itself.

Parameters

- **prefix** (*str*) – prefix to prepend to all parameter names.
- **recurse** (*bool*) – if True, then yields parameters of this module and all submodules. Otherwise, yields only parameters that are direct members of this module.

Yields *(str, Parameter)* – Tuple containing the name and parameter

Example:

```
>>> # xdoctest: +SKIP("undefined vars")
>>> for name, param in self.named_parameters():
>>>    if name in ['bias']:
>>>        print(param.size())
```

parameters(*recurse: bool = True*) → Iterator[torch.nn.parameter.Parameter]

Returns an iterator over module parameters.

This is typically passed to an optimizer.

Parameters **recurse** (*bool*) – if True, then yields parameters of this module and all submodules. Otherwise, yields only parameters that are direct members of this module.

Yields *Parameter* – module parameter

Example:

```
>>> # xdoctest: +SKIP("undefined vars")
>>> for param in model.parameters():
>>>     print(type(param), param.size())
<class 'torch.Tensor'> (20L,)
<class 'torch.Tensor'> (20L, 1L, 5L, 5L)
```

register_backward_hook(*hook: Callable[[torch.nn.modules.module.Module, Union[Tuple[torch.Tensor, ...], torch.Tensor], Union[Tuple[torch.Tensor, ...], torch.Tensor]], Union[None, torch.Tensor]]*) → torch.utils.hooks.RemovableHandle

Registers a backward hook on the module.

This function is deprecated in favor of `register_full_backward_hook()` and the behavior of this function will change in future versions.

Returns a handle that can be used to remove the added hook by calling `handle.remove()`

Return type `torch.utils.hooks.RemovableHandle`

register_buffer(*name: str*, *tensor: Optional[torch.Tensor]*, *persistent: bool = True*) → None

Adds a buffer to the module.

This is typically used to register a buffer that should not to be considered a model parameter. For example, BatchNorm's `running_mean` is not a parameter, but is part of the module's state. Buffers, by default, are persistent and will be saved alongside parameters. This behavior can be changed by setting `persistent` to `False`. The only difference between a persistent buffer and a non-persistent buffer is that the latter will not be a part of this module's *state_dict*.

Buffers can be accessed as attributes using given names.

Parameters

- **name** (*str*) – name of the buffer. The buffer can be accessed from this module using the given name

- **tensor** (*Tensor or None*) – buffer to be registered. If None, then operations that run on buffers, such as *cuda*, are ignored. If None, the buffer is **not** included in the module's *state_dict*.
- **persistent** (*bool*) – whether the buffer is part of this module's *state_dict*.

Example:

```
>>> # xdoctest: +SKIP("undefined vars")
>>> self.register_buffer('running_mean', torch.zeros(num_features))
```

register_forward_hook(*hook: Callable[[...], None]*) → torch.utils.hooks.RemovableHandle

Registers a forward hook on the module.

The hook will be called every time after *forward()* has computed an output. It should have the following signature:

```
hook(module, input, output) -> None or modified output
```

The input contains only the positional arguments given to the module. Keyword arguments won't be passed to the hooks and only to the forward. The hook can modify the output. It can modify the input inplace but it will not have effect on forward since this is called after *forward()* is called.

Returns a handle that can be used to remove the added hook by calling handle.remove()

Return type torch.utils.hooks.RemovableHandle

register_forward_pre_hook(*hook: Callable[[...], None]*) → torch.utils.hooks.RemovableHandle

Registers a forward pre-hook on the module.

The hook will be called every time before *forward()* is invoked. It should have the following signature:

```
hook(module, input) -> None or modified input
```

The input contains only the positional arguments given to the module. Keyword arguments won't be passed to the hooks and only to the forward. The hook can modify the input. User can either return a tuple or a single modified value in the hook. We will wrap the value into a tuple if a single value is returned(unless that value is already a tuple).

Returns a handle that can be used to remove the added hook by calling handle.remove()

Return type torch.utils.hooks.RemovableHandle

register_full_backward_hook(*hook: Callable[[torch.nn.modules.module.Module, Union[Tuple[torch.Tensor, ...], torch.Tensor], Union[Tuple[torch.Tensor, ...], torch.Tensor]], Union[None, torch.Tensor]]*) → torch.utils.hooks.RemovableHandle

Registers a backward hook on the module.

The hook will be called every time the gradients with respect to a module are computed, i.e. the hook will execute if and only if the gradients with respect to module outputs are computed. The hook should have the following signature:

```
hook(module, grad_input, grad_output) -> tuple(Tensor) or None
```

The grad_input and grad_output are tuples that contain the gradients with respect to the inputs and outputs respectively. The hook should not modify its arguments, but it can optionally return a new gradient with respect to the input that will be used in place of grad_input in subsequent computations. grad_input will only correspond to the inputs given as positional arguments and all kwarg arguments are ignored. Entries in grad_input and grad_output will be None for all non-Tensor arguments.

For technical reasons, when this hook is applied to a Module, its forward function will receive a view of each Tensor passed to the Module. Similarly the caller will receive a view of each Tensor returned by the Module's forward function.

Warning: Modifying inputs or outputs inplace is not allowed when using backward hooks and will raise an error.

Returns a handle that can be used to remove the added hook by calling `handle.remove()`

Return type `torch.utils.hooks.RemovableHandle`

register_load_state_dict_post_hook(*hook*)

Registers a post hook to be run after module's `load_state_dict` is called.

It should have the following signature:: hook(module, incompatible_keys) -> None

The `module` argument is the current module that this hook is registered on, and the `incompatible_keys` argument is a `NamedTuple` consisting of attributes `missing_keys` and `unexpected_keys`. `missing_keys` is a `list` of `str` containing the missing keys and `unexpected_keys` is a `list` of `str` containing the unexpected keys.

The given incompatible_keys can be modified inplace if needed.

Note that the checks performed when calling *load_state_dict()* with `strict=True` are affected by modifications the hook makes to `missing_keys` or `unexpected_keys`, as expected. Additions to either set of keys will result in an error being thrown when `strict=True`, and clearning out both missing and unexpected keys will avoid an error.

Returns a handle that can be used to remove the added hook by calling `handle.remove()`

Return type `torch.utils.hooks.RemovableHandle`

register_module(*name: str*, *module: Optional[torch.nn.modules.module.Module]*) → None

Alias for *add_module()*.

register_parameter(*name: str*, *param: Optional[torch.nn.parameter.Parameter]*) → None

Adds a parameter to the module.

The parameter can be accessed as an attribute using given name.

Parameters

- **name** (*str*) – name of the parameter. The parameter can be accessed from this module using the given name
- **param** (*Parameter or None*) – parameter to be added to the module. If `None`, then operations that run on parameters, such as *cuda*, are ignored. If `None`, the parameter is **not** included in the module's *state_dict*.

requires_grad_(*requires_grad: bool = True*) → torch.nn.modules.module.T

Change if autograd should record operations on parameters in this module.

This method sets the parameters' `requires_grad` attributes in-place.

This method is helpful for freezing part of the module for finetuning or training parts of a model individually (e.g., GAN training).

See locally-disable-grad-doc for a comparison between *.requires_grad_()* and several similar mechanisms that may be confused with it.

Parameters **requires_grad** (*bool*) – whether autograd should record operations on parameters in this module. Default: `True`.

Returns self

Return type Module

set_extra_state(*state: Any*)
This function is called from *load_state_dict()* to handle any extra state found within the *state_dict*. Implement this function and a corresponding *get_extra_state()* for your module if you need to store extra state within its *state_dict*.

Parameters **state** (*dict*) – Extra state from the *state_dict*

share_memory() → torch.nn.modules.module.T
See `torch.Tensor.share_memory_()`

state_dict(**args*, *destination=None*, *prefix=''*, *keep_vars=False*)
Returns a dictionary containing references to the whole state of the module.

Both parameters and persistent buffers (e.g. running averages) are included. Keys are corresponding parameter and buffer names. Parameters and buffers set to `None` are not included.

Note: The returned object is a shallow copy. It contains references to the module's parameters and buffers.

Warning: Currently `state_dict()` also accepts positional arguments for `destination`, `prefix` and `keep_vars` in order. However, this is being deprecated and keyword arguments will be enforced in future releases.

Warning: Please avoid the use of argument `destination` as it is not designed for end-users.

Parameters

- **destination** (*dict, optional*) – If provided, the state of module will be updated into the dict and the same object is returned. Otherwise, an `OrderedDict` will be created and returned. Default: `None`.
- **prefix** (*str, optional*) – a prefix added to parameter and buffer names to compose the keys in state_dict. Default: `''`.
- **keep_vars** (*bool, optional*) – by default the `Tensor` s returned in the state dict are detached from autograd. If it's set to `True`, detaching will not be performed. Default: `False`.

Returns a dictionary containing a whole state of the module

Return type dict

Example:

```
>>> # xdoctest: +SKIP("undefined vars")
>>> module.state_dict().keys()
['bias', 'weight']
```

to(**args*, ***kwargs*)
Moves and/or casts the parameters and buffers.

This can be called as

to(*device=None*, *dtype=None*, *non_blocking=False*)

to(*dtype*, *non_blocking=False*)

to(*tensor*, *non_blocking=False*)

to(*memory_format=torch.channels_last*)

Its signature is similar to `torch.Tensor.to()`, but only accepts floating point or complex `dtype`s. In addition, this method will only cast the floating point or complex parameters and buffers to `dtype` (if given). The integral parameters and buffers will be moved `device`, if that is given, but with dtypes unchanged. When `non_blocking` is set, it tries to convert/move asynchronously with respect to the host if possible, e.g., moving CPU Tensors with pinned memory to CUDA devices.

See below for examples.

Note: This method modifies the module in-place.

Parameters

- **device** (`torch.device`) – the desired device of the parameters and buffers in this module
- **dtype** (`torch.dtype`) – the desired floating point or complex dtype of the parameters and buffers in this module
- **tensor** (*torch.Tensor*) – Tensor whose dtype and device are the desired dtype and device for all parameters and buffers in this module
- **memory_format** (`torch.memory_format`) – the desired memory format for 4D parameters and buffers in this module (keyword only argument)

Returns self

Return type Module

Examples:

```
>>> # xdoctest: +IGNORE_WANT("non-deterministic")
>>> linear = nn.Linear(2, 2)
>>> linear.weight
Parameter containing:
tensor([[ 0.1913, -0.3420],
        [-0.5113, -0.2325]])
>>> linear.to(torch.double)
Linear(in_features=2, out_features=2, bias=True)
>>> linear.weight
Parameter containing:
tensor([[ 0.1913, -0.3420],
        [-0.5113, -0.2325]], dtype=torch.float64)
>>> # xdoctest: +REQUIRES(env:TORCH_DOCTEST_CUDA1)
>>> gpu1 = torch.device("cuda:1")
>>> linear.to(gpu1, dtype=torch.half, non_blocking=True)
Linear(in_features=2, out_features=2, bias=True)
>>> linear.weight
Parameter containing:
tensor([[ 0.1914, -0.3420],
```

(continues on next page)

(continued from previous page)

```
        [-0.5112, -0.2324]], dtype=torch.float16, device='cuda:1')
>>> cpu = torch.device("cpu")
>>> linear.to(cpu)
Linear(in_features=2, out_features=2, bias=True)
>>> linear.weight
Parameter containing:
tensor([[ 0.1914, -0.3420],
        [-0.5112, -0.2324]], dtype=torch.float16)

>>> linear = nn.Linear(2, 2, bias=None).to(torch.cdouble)
>>> linear.weight
Parameter containing:
tensor([[ 0.3741+0.j,  0.2382+0.j],
        [ 0.5593+0.j, -0.4443+0.j]], dtype=torch.complex128)
>>> linear(torch.ones(3, 2, dtype=torch.cdouble))
tensor([[0.6122+0.j, 0.1150+0.j],
        [0.6122+0.j, 0.1150+0.j],
        [0.6122+0.j, 0.1150+0.j]], dtype=torch.complex128)
```

to_empty(*, *device: Union[str, torch.device]*) → torch.nn.modules.module.T
Moves the parameters and buffers to the specified device without copying storage.

Parameters **device** (`torch.device`) – The desired device of the parameters and buffers in this module.

Returns self

Return type Module

train(*mode: bool = True*) → torch.nn.modules.module.T
Sets the module in training mode.

This has any effect only on certain modules. See documentations of particular modules for details of their behaviors in training/evaluation mode, if they are affected, e.g. `Dropout`, `BatchNorm`, etc.

Parameters **mode** (*bool*) – whether to set training mode (`True`) or evaluation mode (`False`). Default: `True`.

Returns self

Return type Module

training: bool

type(*dst_type: Union[torch.dtype, str]*) → torch.nn.modules.module.T
Casts all parameters and buffers to `dst_type`.

Note: This method modifies the module in-place.

Parameters **dst_type** (*type or string*) – the desired type

Returns self

Return type Module

xpu(*device: Optional[Union[int, torch.device]] = None*) → torch.nn.modules.module.T
Moves all model parameters and buffers to the XPU.

This also makes associated parameters and buffers different objects. So it should be called before constructing optimizer if the module will live on XPU while being optimized.

Note: This method modifies the module in-place.

Parameters **device** (*int, optional*) – if specified, all parameters will be copied to that device

Returns self

Return type Module

zero_grad(*set_to_none: bool = False*) → None

Sets gradients of all model parameters to zero. See similar function under `torch.optim.Optimizer` for more context.

Parameters **set_to_none** (*bool*) – instead of setting to zero, set the grads to None. See `torch.optim.Optimizer.zero_grad()` for details.

create_alternating_binary_mask(*features: int*, *even: bool = True*) → torch.Tensor

Creates a binary mask of a given dimension which alternates its masking. :param features: Dimension of mask. :param even: If True, even values are assigned 1s, odd 0s. If False, vice versa. :return: Alternating binary mask of type torch.Tensor.

synthcity.plugins.core.models.vae module

class Decoder(*n_units_embedding: int*, *n_units_out: int*, *n_layers_hidden: int = 1*, *n_units_hidden: int = 100*, *nonlin: str = 'relu'*, *nonlin_out: Optional[List[Tuple[str, int]]] = None*, *random_state: int = 0*, *dropout: float = 0.1*, *batch_norm: bool = True*, *residual: bool = False*, *device: Any = device(type='cpu')*)

Bases: `torch.nn.modules.module.Module`

T_destination

alias of TypeVar('T_destination', bound=`Dict[str, Any]`)

add_module(*name: str*, *module: Optional[torch.nn.modules.module.Module]*) → None

Adds a child module to the current module.

The module can be accessed as an attribute using the given name.

Parameters

- **name** (*str*) – name of the child module. The child module can be accessed from this module using the given name
- **module** (*Module*) – child module to be added to the module.

apply(*fn: Callable[[torch.nn.modules.module.Module], None]*) → torch.nn.modules.module.T

Applies `fn` recursively to every submodule (as returned by `.children()`) as well as self. Typical use includes initializing the parameters of a model (see also nn-init-doc).

Parameters **fn** (`Module` -> None) – function to be applied to each submodule

Returns self

Return type Module

Example:

```
>>> @torch.no_grad()
>>> def init_weights(m):
>>>     print(m)
>>>     if type(m) == nn.Linear:
>>>         m.weight.fill_(1.0)
>>>         print(m.weight)
>>> net = nn.Sequential(nn.Linear(2, 2), nn.Linear(2, 2))
>>> net.apply(init_weights)
Linear(in_features=2, out_features=2, bias=True)
Parameter containing:
tensor([[1., 1.],
        [1., 1.]], requires_grad=True)
Linear(in_features=2, out_features=2, bias=True)
Parameter containing:
tensor([[1., 1.],
        [1., 1.]], requires_grad=True)
Sequential(
  (0): Linear(in_features=2, out_features=2, bias=True)
  (1): Linear(in_features=2, out_features=2, bias=True)
)
```

bfloat16() → torch.nn.modules.module.T
Casts all floating point parameters and buffers to `bfloat16` datatype.

Note: This method modifies the module in-place.

Returns self

Return type Module

buffers(*recurse: bool = True*) → Iterator[torch.Tensor]
Returns an iterator over module buffers.

Parameters **recurse** (*bool*) – if True, then yields buffers of this module and all submodules. Otherwise, yields only buffers that are direct members of this module.

Yields *torch.Tensor* – module buffer

Example:

```
>>> # xdoctest: +SKIP("undefined vars")
>>> for buf in model.buffers():
>>>     print(type(buf), buf.size())
<class 'torch.Tensor'> (20L,)
<class 'torch.Tensor'> (20L, 1L, 5L, 5L)
```

children() → Iterator[torch.nn.modules.module.Module]
Returns an iterator over immediate children modules.

Yields *Module* – a child module

cpu() → torch.nn.modules.module.T
Moves all model parameters and buffers to the CPU.

Note: This method modifies the module in-place.

Returns self

Return type Module

cuda(*device: Optional[Union[int, torch.device]] = None*) → torch.nn.modules.module.T
Moves all model parameters and buffers to the GPU.

This also makes associated parameters and buffers different objects. So it should be called before constructing optimizer if the module will live on GPU while being optimized.

Note: This method modifies the module in-place.

Parameters **device** (*int, optional*) – if specified, all parameters will be copied to that device

Returns self

Return type Module

double() → torch.nn.modules.module.T
Casts all floating point parameters and buffers to `double` datatype.

Note: This method modifies the module in-place.

Returns self

Return type Module

dump_patches: bool = False

eval() → torch.nn.modules.module.T
Sets the module in evaluation mode.

This has any effect only on certain modules. See documentations of particular modules for details of their behaviors in training/evaluation mode, if they are affected, e.g. `Dropout`, `BatchNorm`, etc.

This is equivalent with `self.train(False)`.

See locally-disable-grad-doc for a comparison between *.eval()* and several similar mechanisms that may be confused with it.

Returns self

Return type Module

extra_repr() → str
Set the extra representation of the module

To print customized extra information, you should re-implement this method in your own modules. Both single-line and multi-line strings are acceptable.

float() → torch.nn.modules.module.T
Casts all floating point parameters and buffers to `float` datatype.

Note: This method modifies the module in-place.

Returns self

Return type Module

forward(*X: torch.Tensor*, *cond: Optional[torch.Tensor] = None*) → torch.Tensor
Defines the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

get_buffer(*target: str*) → torch.Tensor
Returns the buffer given by `target` if it exists, otherwise throws an error.

See the docstring for `get_submodule` for a more detailed explanation of this method's functionality as well as how to correctly specify `target`.

Parameters **target** – The fully-qualified string name of the buffer to look for. (See `get_submodule` for how to specify a fully-qualified string.)

Returns The buffer referenced by `target`

Return type torch.Tensor

Raises **AttributeError** – If the target string references an invalid path or resolves to something that is not a buffer

get_extra_state() → Any
Returns any extra state to include in the module's state_dict. Implement this and a corresponding *set_extra_state()* for your module if you need to store extra state. This function is called when building the module's *state_dict()*.

Note that extra state should be pickleable to ensure working serialization of the state_dict. We only provide provide backwards compatibility guarantees for serializing Tensors; other objects may break backwards compatibility if their serialized pickled form changes.

Returns Any extra state to store in the module's state_dict

Return type object

get_parameter(*target: str*) → torch.nn.parameter.Parameter
Returns the parameter given by `target` if it exists, otherwise throws an error.

See the docstring for `get_submodule` for a more detailed explanation of this method's functionality as well as how to correctly specify `target`.

Parameters **target** – The fully-qualified string name of the Parameter to look for. (See `get_submodule` for how to specify a fully-qualified string.)

Returns The Parameter referenced by `target`

Return type torch.nn.Parameter

Raises **AttributeError** – If the target string references an invalid path or resolves to something that is not an `nn.Parameter`

get_submodule(*target: str*) → torch.nn.modules.module.Module
Returns the submodule given by `target` if it exists, otherwise throws an error.

For example, let's say you have an `nn.Module` `A` that looks like this:

```
A(
    (net_b): Module(
        (net_c): Module(
            (conv): Conv2d(16, 33, kernel_size=(3, 3), stride=(2, 2))
        )
        (linear): Linear(in_features=100, out_features=200, bias=True)
    )
)
```

(The diagram shows an `nn.Module` `A`. `A` has a nested submodule `net_b`, which itself has two submodules `net_c` and `linear`. `net_c` then has a submodule `conv`.)

To check whether or not we have the `linear` submodule, we would call `get_submodule("net_b.linear")`. To check whether we have the `conv` submodule, we would call `get_submodule("net_b.net_c.conv")`.

The runtime of `get_submodule` is bounded by the degree of module nesting in `target`. A query against `named_modules` achieves the same result, but it is O(N) in the number of transitive modules. So, for a simple check to see if some submodule exists, `get_submodule` should always be used.

Parameters `target` – The fully-qualified string name of the submodule to look for. (See above example for how to specify a fully-qualified string.)

Returns The submodule referenced by `target`

Return type torch.nn.Module

Raises `AttributeError` – If the target string references an invalid path or resolves to something that is not an `nn.Module`

half() → torch.nn.modules.module.T
Casts all floating point parameters and buffers to `half` datatype.

Note: This method modifies the module in-place.

Returns self

Return type Module

ipu(*device: Optional[Union[int, torch.device]] = None*) → torch.nn.modules.module.T
Moves all model parameters and buffers to the IPU.

This also makes associated parameters and buffers different objects. So it should be called before constructing optimizer if the module will live on IPU while being optimized.

Note: This method modifies the module in-place.

Parameters `device` (*int, optional*) – if specified, all parameters will be copied to that device

Returns self

Return type Module

load_state_dict(*state_dict: Mapping[str, Any]*, *strict: bool = True*)
Copies parameters and buffers from *state_dict* into this module and its descendants. If `strict` is `True`, then the keys of *state_dict* must exactly match the keys returned by this module's `state_dict()` function.

Parameters

- **state_dict** (*dict*) – a dict containing parameters and persistent buffers.
- **strict** (*bool, optional*) – whether to strictly enforce that the keys in *state_dict* match the keys returned by this module's `state_dict()` function. Default: `True`

Returns

- **missing_keys** is a list of str containing the missing keys
- **unexpected_keys** is a list of str containing the unexpected keys

Return type `NamedTuple` with `missing_keys` and `unexpected_keys` fields

Note: If a parameter or buffer is registered as `None` and its corresponding key exists in *state_dict*, *load_state_dict()* will raise a `RuntimeError`.

---

modules() → Iterator[torch.nn.modules.module.Module]
Returns an iterator over all modules in the network.

Yields *Module* – a module in the network

Note: Duplicate modules are returned only once. In the following example, `l` will be returned only once.

Example:

```
>>> l = nn.Linear(2, 2)
>>> net = nn.Sequential(l, l)
>>> for idx, m in enumerate(net.modules()):
...     print(idx, '->', m)

0 -> Sequential(
  (0): Linear(in_features=2, out_features=2, bias=True)
  (1): Linear(in_features=2, out_features=2, bias=True)
)
1 -> Linear(in_features=2, out_features=2, bias=True)
```

named_buffers(*prefix: str = ''*, *recurse: bool = True*) → Iterator[Tuple[str, torch.Tensor]]
Returns an iterator over module buffers, yielding both the name of the buffer as well as the buffer itself.

Parameters

- **prefix** (*str*) – prefix to prepend to all buffer names.
- **recurse** (*bool*) – if True, then yields buffers of this module and all submodules. Otherwise, yields only buffers that are direct members of this module.

Yields *(str, torch.Tensor)* – Tuple containing the name and buffer

Example:

```
>>> # xdoctest: +SKIP("undefined vars")
>>> for name, buf in self.named_buffers():
>>>    if name in ['running_var']:
>>>        print(buf.size())
```

named_children() → Iterator[Tuple[str, torch.nn.modules.module.Module]]

Returns an iterator over immediate children modules, yielding both the name of the module as well as the module itself.

Yields *(str, Module)* – Tuple containing a name and child module

Example:

```
>>> # xdoctest: +SKIP("undefined vars")
>>> for name, module in model.named_children():
>>>     if name in ['conv4', 'conv5']:
>>>         print(module)
```

named_modules(*memo: Optional[Set[torch.nn.modules.module.Module]] = None*, *prefix: str = ''*, *remove_duplicate: bool = True*)

Returns an iterator over all modules in the network, yielding both the name of the module as well as the module itself.

Parameters

- **memo** – a memo to store the set of modules already added to the result
- **prefix** – a prefix that will be added to the name of the module
- **remove_duplicate** – whether to remove the duplicated module instances in the result or not

Yields *(str, Module)* – Tuple of name and module

Note: Duplicate modules are returned only once. In the following example, `l` will be returned only once.

Example:

```
>>> l = nn.Linear(2, 2)
>>> net = nn.Sequential(l, l)
>>> for idx, m in enumerate(net.named_modules()):
...     print(idx, '->', m)

0 -> ('', Sequential(
  (0): Linear(in_features=2, out_features=2, bias=True)
  (1): Linear(in_features=2, out_features=2, bias=True)
))
1 -> ('0', Linear(in_features=2, out_features=2, bias=True))
```

named_parameters(*prefix: str = ''*, *recurse: bool = True*) → Iterator[Tuple[str, torch.nn.parameter.Parameter]]

Returns an iterator over module parameters, yielding both the name of the parameter as well as the parameter itself.

Parameters

- **prefix** (*str*) – prefix to prepend to all parameter names.

- **recurse** (*bool*) – if True, then yields parameters of this module and all submodules. Otherwise, yields only parameters that are direct members of this module.

Yields *(str, Parameter)* – Tuple containing the name and parameter

Example:

```
>>> # xdoctest: +SKIP("undefined vars")
>>> for name, param in self.named_parameters():
>>>    if name in ['bias']:
>>>        print(param.size())
```

parameters(*recurse: bool = True*) → Iterator[torch.nn.parameter.Parameter]

Returns an iterator over module parameters.

This is typically passed to an optimizer.

Parameters **recurse** (*bool*) – if True, then yields parameters of this module and all submodules. Otherwise, yields only parameters that are direct members of this module.

Yields *Parameter* – module parameter

Example:

```
>>> # xdoctest: +SKIP("undefined vars")
>>> for param in model.parameters():
>>>     print(type(param), param.size())
<class 'torch.Tensor'> (20L,)
<class 'torch.Tensor'> (20L, 1L, 5L, 5L)
```

register_backward_hook(*hook: Callable[[torch.nn.modules.module.Module, Union[Tuple[torch.Tensor, ...], torch.Tensor], Union[Tuple[torch.Tensor, ...], torch.Tensor]], Union[None, torch.Tensor]]*) → torch.utils.hooks.RemovableHandle

Registers a backward hook on the module.

This function is deprecated in favor of `register_full_backward_hook()` and the behavior of this function will change in future versions.

Returns a handle that can be used to remove the added hook by calling `handle.remove()`

Return type `torch.utils.hooks.RemovableHandle`

register_buffer(*name: str*, *tensor: Optional[torch.Tensor]*, *persistent: bool = True*) → None

Adds a buffer to the module.

This is typically used to register a buffer that should not to be considered a model parameter. For example, BatchNorm's `running_mean` is not a parameter, but is part of the module's state. Buffers, by default, are persistent and will be saved alongside parameters. This behavior can be changed by setting `persistent` to `False`. The only difference between a persistent buffer and a non-persistent buffer is that the latter will not be a part of this module's *state_dict*.

Buffers can be accessed as attributes using given names.

Parameters

- **name** (*str*) – name of the buffer. The buffer can be accessed from this module using the given name
- **tensor** (*Tensor or None*) – buffer to be registered. If `None`, then operations that run on buffers, such as *cuda*, are ignored. If `None`, the buffer is **not** included in the module's *state_dict*.

- **persistent** (*bool*) – whether the buffer is part of this module's *state_dict*.

Example:

```
>>> # xdoctest: +SKIP("undefined vars")
>>> self.register_buffer('running_mean', torch.zeros(num_features))
```

register_forward_hook(*hook: Callable[[...], None]*) → torch.utils.hooks.RemovableHandle

Registers a forward hook on the module.

The hook will be called every time after *forward()* has computed an output. It should have the following signature:

```
hook(module, input, output) -> None or modified output
```

The input contains only the positional arguments given to the module. Keyword arguments won't be passed to the hooks and only to the `forward`. The hook can modify the output. It can modify the input inplace but it will not have effect on forward since this is called after *forward()* is called.

Returns a handle that can be used to remove the added hook by calling `handle.remove()`

Return type `torch.utils.hooks.RemovableHandle`

register_forward_pre_hook(*hook: Callable[[...], None]*) → torch.utils.hooks.RemovableHandle

Registers a forward pre-hook on the module.

The hook will be called every time before *forward()* is invoked. It should have the following signature:

```
hook(module, input) -> None or modified input
```

The input contains only the positional arguments given to the module. Keyword arguments won't be passed to the hooks and only to the `forward`. The hook can modify the input. User can either return a tuple or a single modified value in the hook. We will wrap the value into a tuple if a single value is returned(unless that value is already a tuple).

Returns a handle that can be used to remove the added hook by calling `handle.remove()`

Return type `torch.utils.hooks.RemovableHandle`

register_full_backward_hook(*hook: Callable[[torch.nn.modules.module.Module, Union[Tuple[torch.Tensor, ...], torch.Tensor], Union[Tuple[torch.Tensor, ...], torch.Tensor]], Union[None, torch.Tensor]]*) → torch.utils.hooks.RemovableHandle

Registers a backward hook on the module.

The hook will be called every time the gradients with respect to a module are computed, i.e. the hook will execute if and only if the gradients with respect to module outputs are computed. The hook should have the following signature:

```
hook(module, grad_input, grad_output) -> tuple(Tensor) or None
```

The `grad_input` and `grad_output` are tuples that contain the gradients with respect to the inputs and outputs respectively. The hook should not modify its arguments, but it can optionally return a new gradient with respect to the input that will be used in place of `grad_input` in subsequent computations. `grad_input` will only correspond to the inputs given as positional arguments and all kwarg arguments are ignored. Entries in `grad_input` and `grad_output` will be `None` for all non-Tensor arguments.

For technical reasons, when this hook is applied to a Module, its forward function will receive a view of each Tensor passed to the Module. Similarly the caller will receive a view of each Tensor returned by the Module's forward function.

> **Warning:** Modifying inputs or outputs inplace is not allowed when using backward hooks and will raise an error.

Returns a handle that can be used to remove the added hook by calling `handle.remove()`

Return type `torch.utils.hooks.RemovableHandle`

register_load_state_dict_post_hook(*hook*)

Registers a post hook to be run after module's `load_state_dict` is called.

It should have the following signature:: hook(module, incompatible_keys) -> None

The `module` argument is the current module that this hook is registered on, and the `incompatible_keys` argument is a `NamedTuple` consisting of attributes `missing_keys` and `unexpected_keys`. `missing_keys` is a `list` of `str` containing the missing keys and `unexpected_keys` is a `list` of `str` containing the unexpected keys.

The given incompatible_keys can be modified inplace if needed.

Note that the checks performed when calling *load_state_dict()* with `strict=True` are affected by modifications the hook makes to `missing_keys` or `unexpected_keys`, as expected. Additions to either set of keys will result in an error being thrown when `strict=True`, and clearning out both missing and unexpected keys will avoid an error.

Returns a handle that can be used to remove the added hook by calling `handle.remove()`

Return type `torch.utils.hooks.RemovableHandle`

register_module(*name: str*, *module: Optional[torch.nn.modules.module.Module]*) → None

Alias for *add_module()*.

register_parameter(*name: str*, *param: Optional[torch.nn.parameter.Parameter]*) → None

Adds a parameter to the module.

The parameter can be accessed as an attribute using given name.

Parameters

- **name** (*str*) – name of the parameter. The parameter can be accessed from this module using the given name
- **param** (*Parameter or None*) – parameter to be added to the module. If `None`, then operations that run on parameters, such as *cuda*, are ignored. If `None`, the parameter is **not** included in the module's *state_dict*.

requires_grad_(*requires_grad: bool = True*) → torch.nn.modules.module.T

Change if autograd should record operations on parameters in this module.

This method sets the parameters' `requires_grad` attributes in-place.

This method is helpful for freezing part of the module for finetuning or training parts of a model individually (e.g., GAN training).

See locally-disable-grad-doc for a comparison between *.requires_grad_()* and several similar mechanisms that may be confused with it.

Parameters **requires_grad** (*bool*) – whether autograd should record operations on parameters in this module. Default: `True`.

Returns self

Return type Module

set_extra_state(*state: Any*)
This function is called from *load_state_dict()* to handle any extra state found within the *state_dict*. Implement this function and a corresponding *get_extra_state()* for your module if you need to store extra state within its *state_dict*.

Parameters **state** (*dict*) – Extra state from the *state_dict*

share_memory() → torch.nn.modules.module.T
See `torch.Tensor.share_memory_()`

state_dict(**args*, *destination=None*, *prefix=''*, *keep_vars=False*)
Returns a dictionary containing references to the whole state of the module.

Both parameters and persistent buffers (e.g. running averages) are included. Keys are corresponding parameter and buffer names. Parameters and buffers set to `None` are not included.

Note: The returned object is a shallow copy. It contains references to the module's parameters and buffers.

Warning: Currently `state_dict()` also accepts positional arguments for `destination`, `prefix` and `keep_vars` in order. However, this is being deprecated and keyword arguments will be enforced in future releases.

Warning: Please avoid the use of argument `destination` as it is not designed for end-users.

Parameters

- **destination** (*dict, optional*) – If provided, the state of module will be updated into the dict and the same object is returned. Otherwise, an `OrderedDict` will be created and returned. Default: `None`.
- **prefix** (*str, optional*) – a prefix added to parameter and buffer names to compose the keys in state_dict. Default: `''`.
- **keep_vars** (*bool, optional*) – by default the `Tensor` s returned in the state dict are detached from autograd. If it's set to `True`, detaching will not be performed. Default: `False`.

Returns a dictionary containing a whole state of the module

Return type dict

Example:

```
>>> # xdoctest: +SKIP("undefined vars")
>>> module.state_dict().keys()
['bias', 'weight']
```

to(**args*, ***kwargs*)
Moves and/or casts the parameters and buffers.

This can be called as

to(*device=None*, *dtype=None*, *non_blocking=False*)

to(*dtype*, *non_blocking=False*)

to(*tensor*, *non_blocking=False*)

to(*memory_format=torch.channels_last*)

Its signature is similar to `torch.Tensor.to()`, but only accepts floating point or complex `dtype`s. In addition, this method will only cast the floating point or complex parameters and buffers to `dtype` (if given). The integral parameters and buffers will be moved `device`, if that is given, but with dtypes unchanged. When `non_blocking` is set, it tries to convert/move asynchronously with respect to the host if possible, e.g., moving CPU Tensors with pinned memory to CUDA devices.

See below for examples.

Note: This method modifies the module in-place.

Parameters

- **device** (`torch.device`) – the desired device of the parameters and buffers in this module
- **dtype** (`torch.dtype`) – the desired floating point or complex dtype of the parameters and buffers in this module
- **tensor** (*torch.Tensor*) – Tensor whose dtype and device are the desired dtype and device for all parameters and buffers in this module
- **memory_format** (`torch.memory_format`) – the desired memory format for 4D parameters and buffers in this module (keyword only argument)

Returns self

Return type Module

Examples:

```
>>> # xdoctest: +IGNORE_WANT("non-deterministic")
>>> linear = nn.Linear(2, 2)
>>> linear.weight
Parameter containing:
tensor([[ 0.1913, -0.3420],
        [-0.5113, -0.2325]])
>>> linear.to(torch.double)
Linear(in_features=2, out_features=2, bias=True)
>>> linear.weight
Parameter containing:
tensor([[ 0.1913, -0.3420],
        [-0.5113, -0.2325]], dtype=torch.float64)
>>> # xdoctest: +REQUIRES(env:TORCH_DOCTEST_CUDA1)
>>> gpu1 = torch.device("cuda:1")
>>> linear.to(gpu1, dtype=torch.half, non_blocking=True)
Linear(in_features=2, out_features=2, bias=True)
>>> linear.weight
Parameter containing:
tensor([[ 0.1914, -0.3420],
        [-0.5112, -0.2324]], dtype=torch.float16, device='cuda:1')
>>> cpu = torch.device("cpu")
>>> linear.to(cpu)
```

(continues on next page)

(continued from previous page)

```
Linear(in_features=2, out_features=2, bias=True)
>>> linear.weight
Parameter containing:
tensor([[ 0.1914, -0.3420],
        [-0.5112, -0.2324]], dtype=torch.float16)

>>> linear = nn.Linear(2, 2, bias=None).to(torch.cdouble)
>>> linear.weight
Parameter containing:
tensor([[ 0.3741+0.j,  0.2382+0.j],
        [ 0.5593+0.j, -0.4443+0.j]], dtype=torch.complex128)
>>> linear(torch.ones(3, 2, dtype=torch.cdouble))
tensor([[0.6122+0.j, 0.1150+0.j],
        [0.6122+0.j, 0.1150+0.j],
        [0.6122+0.j, 0.1150+0.j]], dtype=torch.complex128)
```

to_empty(*, *device: Union[str, torch.device]*) → torch.nn.modules.module.T
Moves the parameters and buffers to the specified device without copying storage.

Parameters **device** (`torch.device`) – The desired device of the parameters and buffers in this module.

Returns self

Return type Module

train(*mode: bool = True*) → torch.nn.modules.module.T
Sets the module in training mode.

This has any effect only on certain modules. See documentations of particular modules for details of their behaviors in training/evaluation mode, if they are affected, e.g. `Dropout`, `BatchNorm`, etc.

Parameters **mode** (*bool*) – whether to set training mode (`True`) or evaluation mode (`False`). Default: `True`.

Returns self

Return type Module

training: bool

type(*dst_type: Union[torch.dtype, str]*) → torch.nn.modules.module.T
Casts all parameters and buffers to `dst_type`.

Note: This method modifies the module in-place.

Parameters **dst_type** (*type or string*) – the desired type

Returns self

Return type Module

xpu(*device: Optional[Union[int, torch.device]] = None*) → torch.nn.modules.module.T
Moves all model parameters and buffers to the XPU.

This also makes associated parameters and buffers different objects. So it should be called before constructing optimizer if the module will live on XPU while being optimized.

Note: This method modifies the module in-place.

Parameters **device** (*int, optional*) – if specified, all parameters will be copied to that device

Returns self

Return type Module

zero_grad(*set_to_none: bool = False*) → None
Sets gradients of all model parameters to zero. See similar function under `torch.optim.Optimizer` for more context.

Parameters **set_to_none** (*bool*) – instead of setting to zero, set the grads to None. See `torch.optim.Optimizer.zero_grad()` for details.

class Encoder(*n_units_in: int*, *n_units_embedding: int*, *n_layers_hidden: int = 1*, *n_units_hidden: int = 100*, *nonlin: str = 'relu'*, *random_state: int = 0*, *dropout: float = 0.1*, *batch_norm: bool = True*, *residual: bool = False*, *device: Any = device(type='cpu')*)
Bases: `torch.nn.modules.module.Module`

T_destination
alias of TypeVar('T_destination', bound=`Dict[str, Any]`)

add_module(*name: str*, *module: Optional[torch.nn.modules.module.Module]*) → None
Adds a child module to the current module.

The module can be accessed as an attribute using the given name.

Parameters

- **name** (*str*) – name of the child module. The child module can be accessed from this module using the given name
- **module** (*Module*) – child module to be added to the module.

apply(*fn: Callable[[torch.nn.modules.module.Module], None]*) → torch.nn.modules.module.T
Applies `fn` recursively to every submodule (as returned by `.children()`) as well as self. Typical use includes initializing the parameters of a model (see also nn-init-doc).

Parameters **fn** (`Module` -> None) – function to be applied to each submodule

Returns self

Return type Module

Example:

```
>>> @torch.no_grad()
>>> def init_weights(m):
>>>     print(m)
>>>     if type(m) == nn.Linear:
>>>         m.weight.fill_(1.0)
>>>         print(m.weight)
>>> net = nn.Sequential(nn.Linear(2, 2), nn.Linear(2, 2))
>>> net.apply(init_weights)
Linear(in_features=2, out_features=2, bias=True)
Parameter containing:
tensor([[1., 1.],
```

(continues on next page)

(continued from previous page)

```
        [1., 1.]], requires_grad=True)
Linear(in_features=2, out_features=2, bias=True)
Parameter containing:
tensor([[1., 1.],
        [1., 1.]], requires_grad=True)
Sequential(
  (0): Linear(in_features=2, out_features=2, bias=True)
  (1): Linear(in_features=2, out_features=2, bias=True)
)
```

bfloat16() → torch.nn.modules.module.T
Casts all floating point parameters and buffers to `bfloat16` datatype.

Note: This method modifies the module in-place.

Returns self

Return type Module

buffers(*recurse: bool = True*) → Iterator[torch.Tensor]
Returns an iterator over module buffers.

Parameters **recurse** (*bool*) – if True, then yields buffers of this module and all submodules. Otherwise, yields only buffers that are direct members of this module.

Yields *torch.Tensor* – module buffer

Example:

```
>>> # xdoctest: +SKIP("undefined vars")
>>> for buf in model.buffers():
>>>     print(type(buf), buf.size())
<class 'torch.Tensor'> (20L,)
<class 'torch.Tensor'> (20L, 1L, 5L, 5L)
```

children() → Iterator[torch.nn.modules.module.Module]
Returns an iterator over immediate children modules.

Yields *Module* – a child module

cpu() → torch.nn.modules.module.T
Moves all model parameters and buffers to the CPU.

Note: This method modifies the module in-place.

Returns self

Return type Module

cuda(*device: Optional[Union[int, torch.device]] = None*) → torch.nn.modules.module.T
Moves all model parameters and buffers to the GPU.

This also makes associated parameters and buffers different objects. So it should be called before constructing optimizer if the module will live on GPU while being optimized.

Note: This method modifies the module in-place.

Parameters **device** (*int, optional*) – if specified, all parameters will be copied to that device

Returns self

Return type Module

double() → torch.nn.modules.module.T
Casts all floating point parameters and buffers to double datatype.

Note: This method modifies the module in-place.

Returns self

Return type Module

dump_patches: bool = False

eval() → torch.nn.modules.module.T
Sets the module in evaluation mode.

This has any effect only on certain modules. See documentations of particular modules for details of their behaviors in training/evaluation mode, if they are affected, e.g. Dropout, BatchNorm, etc.

This is equivalent with self.train(False).

See locally-disable-grad-doc for a comparison between *.eval()* and several similar mechanisms that may be confused with it.

Returns self

Return type Module

extra_repr() → str
Set the extra representation of the module

To print customized extra information, you should re-implement this method in your own modules. Both single-line and multi-line strings are acceptable.

float() → torch.nn.modules.module.T
Casts all floating point parameters and buffers to float datatype.

Note: This method modifies the module in-place.

Returns self

Return type Module

forward(*X: torch.Tensor*, *cond: Optional[torch.Tensor] = None*) → Tuple[torch.Tensor, torch.Tensor]
Defines the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

get_buffer(*target: str*) → torch.Tensor
Returns the buffer given by `target` if it exists, otherwise throws an error.

See the docstring for `get_submodule` for a more detailed explanation of this method's functionality as well as how to correctly specify `target`.

Parameters **target** – The fully-qualified string name of the buffer to look for. (See `get_submodule` for how to specify a fully-qualified string.)

Returns The buffer referenced by `target`

Return type torch.Tensor

Raises **AttributeError** – If the target string references an invalid path or resolves to something that is not a buffer

get_extra_state() → Any
Returns any extra state to include in the module's state_dict. Implement this and a corresponding *set_extra_state()* for your module if you need to store extra state. This function is called when building the module's *state_dict()*.

Note that extra state should be pickleable to ensure working serialization of the state_dict. We only provide provide backwards compatibility guarantees for serializing Tensors; other objects may break backwards compatibility if their serialized pickled form changes.

Returns Any extra state to store in the module's state_dict

Return type object

get_parameter(*target: str*) → torch.nn.parameter.Parameter
Returns the parameter given by `target` if it exists, otherwise throws an error.

See the docstring for `get_submodule` for a more detailed explanation of this method's functionality as well as how to correctly specify `target`.

Parameters **target** – The fully-qualified string name of the Parameter to look for. (See `get_submodule` for how to specify a fully-qualified string.)

Returns The Parameter referenced by `target`

Return type torch.nn.Parameter

Raises **AttributeError** – If the target string references an invalid path or resolves to something that is not an `nn.Parameter`

get_submodule(*target: str*) → torch.nn.modules.module.Module
Returns the submodule given by `target` if it exists, otherwise throws an error.

For example, let's say you have an `nn.Module` `A` that looks like this:

```
A(
    (net_b): Module(
        (net_c): Module(
            (conv): Conv2d(16, 33, kernel_size=(3, 3), stride=(2, 2))
        )
        (linear): Linear(in_features=100, out_features=200, bias=True)
```

(continues on next page)

(continued from previous page)

```
    )
)
```

(The diagram shows an `nn.Module` `A`. `A` has a nested submodule `net_b`, which itself has two submodules `net_c` and `linear`. `net_c` then has a submodule `conv`.)

To check whether or not we have the `linear` submodule, we would call `get_submodule("net_b.linear")`. To check whether we have the `conv` submodule, we would call `get_submodule("net_b.net_c.conv")`.

The runtime of `get_submodule` is bounded by the degree of module nesting in `target`. A query against `named_modules` achieves the same result, but it is O(N) in the number of transitive modules. So, for a simple check to see if some submodule exists, `get_submodule` should always be used.

Parameters `target` – The fully-qualified string name of the submodule to look for. (See above example for how to specify a fully-qualified string.)

Returns The submodule referenced by `target`

Return type torch.nn.Module

Raises `AttributeError` – If the target string references an invalid path or resolves to something that is not an `nn.Module`

`half()` → torch.nn.modules.module.T

Casts all floating point parameters and buffers to `half` datatype.

Note: This method modifies the module in-place.

Returns self

Return type Module

`ipu(`*device: Optional[Union[int, torch.device]] = None*`)` → torch.nn.modules.module.T

Moves all model parameters and buffers to the IPU.

This also makes associated parameters and buffers different objects. So it should be called before constructing optimizer if the module will live on IPU while being optimized.

Note: This method modifies the module in-place.

Parameters `device` (*int, optional*) – if specified, all parameters will be copied to that device

Returns self

Return type Module

`load_state_dict(`*state_dict: Mapping[str, Any]*, *strict: bool = True*`)`

Copies parameters and buffers from *state_dict* into this module and its descendants. If `strict` is `True`, then the keys of *state_dict* must exactly match the keys returned by this module's `state_dict()` function.

Parameters

- `state_dict` (*dict*) – a dict containing parameters and persistent buffers.

- **strict** (*bool, optional*) – whether to strictly enforce that the keys in *state_dict* match the keys returned by this module's `state_dict()` function. Default: `True`

Returns

- **missing_keys** is a list of str containing the missing keys
- **unexpected_keys** is a list of str containing the unexpected keys

Return type `NamedTuple` with `missing_keys` and `unexpected_keys` fields

Note: If a parameter or buffer is registered as `None` and its corresponding key exists in *state_dict*, *load_state_dict()* will raise a `RuntimeError`.

---

modules() → Iterator[torch.nn.modules.module.Module]

Returns an iterator over all modules in the network.

Yields *Module* – a module in the network

Note: Duplicate modules are returned only once. In the following example, `l` will be returned only once.

Example:

```
>>> l = nn.Linear(2, 2)
>>> net = nn.Sequential(l, l)
>>> for idx, m in enumerate(net.modules()):
...     print(idx, '->', m)

0 -> Sequential(
  (0): Linear(in_features=2, out_features=2, bias=True)
  (1): Linear(in_features=2, out_features=2, bias=True)
)
1 -> Linear(in_features=2, out_features=2, bias=True)
```

named_buffers(*prefix: str = ''*, *recurse: bool = True*) → Iterator[Tuple[str, torch.Tensor]]

Returns an iterator over module buffers, yielding both the name of the buffer as well as the buffer itself.

Parameters

- **prefix** (*str*) – prefix to prepend to all buffer names.
- **recurse** (*bool*) – if True, then yields buffers of this module and all submodules. Otherwise, yields only buffers that are direct members of this module.

Yields *(str, torch.Tensor)* – Tuple containing the name and buffer

Example:

```
>>> # xdoctest: +SKIP("undefined vars")
>>> for name, buf in self.named_buffers():
>>>    if name in ['running_var']:
>>>        print(buf.size())
```

named_children() → Iterator[Tuple[str, torch.nn.modules.module.Module]]

Returns an iterator over immediate children modules, yielding both the name of the module as well as the module itself.

Yields *(str, Module)* – Tuple containing a name and child module

Example:

```
>>> # xdoctest: +SKIP("undefined vars")
>>> for name, module in model.named_children():
>>>     if name in ['conv4', 'conv5']:
>>>         print(module)
```

named_modules(*memo: Optional[Set[torch.nn.modules.module.Module]] = None*, *prefix: str = ''*, *remove_duplicate: bool = True*)

Returns an iterator over all modules in the network, yielding both the name of the module as well as the module itself.

Parameters

- **memo** – a memo to store the set of modules already added to the result
- **prefix** – a prefix that will be added to the name of the module
- **remove_duplicate** – whether to remove the duplicated module instances in the result or not

Yields *(str, Module)* – Tuple of name and module

Note: Duplicate modules are returned only once. In the following example, `l` will be returned only once.

Example:

```
>>> l = nn.Linear(2, 2)
>>> net = nn.Sequential(l, l)
>>> for idx, m in enumerate(net.named_modules()):
...     print(idx, '->', m)

0 -> ('', Sequential(
  (0): Linear(in_features=2, out_features=2, bias=True)
  (1): Linear(in_features=2, out_features=2, bias=True)
))
1 -> ('0', Linear(in_features=2, out_features=2, bias=True))
```

named_parameters(*prefix: str = ''*, *recurse: bool = True*) → Iterator[Tuple[str, torch.nn.parameter.Parameter]]

Returns an iterator over module parameters, yielding both the name of the parameter as well as the parameter itself.

Parameters

- **prefix** (*str*) – prefix to prepend to all parameter names.
- **recurse** (*bool*) – if True, then yields parameters of this module and all submodules. Otherwise, yields only parameters that are direct members of this module.

Yields *(str, Parameter)* – Tuple containing the name and parameter

Example:

```
>>> # xdoctest: +SKIP("undefined vars")
>>> for name, param in self.named_parameters():
>>>    if name in ['bias']:
>>>        print(param.size())
```

parameters(*recurse: bool = True*) → Iterator[torch.nn.parameter.Parameter]
Returns an iterator over module parameters.

This is typically passed to an optimizer.

Parameters **recurse** (*bool*) – if True, then yields parameters of this module and all submodules. Otherwise, yields only parameters that are direct members of this module.

Yields *Parameter* – module parameter

Example:

```
>>> # xdoctest: +SKIP("undefined vars")
>>> for param in model.parameters():
>>>     print(type(param), param.size())
<class 'torch.Tensor'> (20L,)
<class 'torch.Tensor'> (20L, 1L, 5L, 5L)
```

register_backward_hook(*hook: Callable[[torch.nn.modules.module.Module, Union[Tuple[torch.Tensor, ...], torch.Tensor], Union[Tuple[torch.Tensor, ...], torch.Tensor]], Union[None, torch.Tensor]]*) → torch.utils.hooks.RemovableHandle
Registers a backward hook on the module.

This function is deprecated in favor of `register_full_backward_hook()` and the behavior of this function will change in future versions.

Returns a handle that can be used to remove the added hook by calling `handle.remove()`

Return type `torch.utils.hooks.RemovableHandle`

register_buffer(*name: str*, *tensor: Optional[torch.Tensor]*, *persistent: bool = True*) → None
Adds a buffer to the module.

This is typically used to register a buffer that should not to be considered a model parameter. For example, BatchNorm's `running_mean` is not a parameter, but is part of the module's state. Buffers, by default, are persistent and will be saved alongside parameters. This behavior can be changed by setting `persistent` to `False`. The only difference between a persistent buffer and a non-persistent buffer is that the latter will not be a part of this module's *state_dict*.

Buffers can be accessed as attributes using given names.

Parameters

- **name** (*str*) – name of the buffer. The buffer can be accessed from this module using the given name
- **tensor** (*Tensor or None*) – buffer to be registered. If `None`, then operations that run on buffers, such as *cuda*, are ignored. If `None`, the buffer is **not** included in the module's *state_dict*.
- **persistent** (*bool*) – whether the buffer is part of this module's *state_dict*.

Example:

```
>>> # xdoctest: +SKIP("undefined vars")
>>> self.register_buffer('running_mean', torch.zeros(num_features))
```

register_forward_hook(*hook: Callable[[...], None]*) → torch.utils.hooks.RemovableHandle
Registers a forward hook on the module.

The hook will be called every time after *forward()* has computed an output. It should have the following signature:

```
hook(module, input, output) -> None or modified output
```

The input contains only the positional arguments given to the module. Keyword arguments won't be passed to the hooks and only to the `forward`. The hook can modify the output. It can modify the input inplace but it will not have effect on forward since this is called after *forward()* is called.

Returns a handle that can be used to remove the added hook by calling `handle.remove()`

Return type `torch.utils.hooks.RemovableHandle`

register_forward_pre_hook(*hook: Callable[[...], None]*) → torch.utils.hooks.RemovableHandle

Registers a forward pre-hook on the module.

The hook will be called every time before *forward()* is invoked. It should have the following signature:

```
hook(module, input) -> None or modified input
```

The input contains only the positional arguments given to the module. Keyword arguments won't be passed to the hooks and only to the `forward`. The hook can modify the input. User can either return a tuple or a single modified value in the hook. We will wrap the value into a tuple if a single value is returned(unless that value is already a tuple).

Returns a handle that can be used to remove the added hook by calling `handle.remove()`

Return type `torch.utils.hooks.RemovableHandle`

register_full_backward_hook(*hook: Callable[[torch.nn.modules.module.Module, Union[Tuple[torch.Tensor, ...], torch.Tensor], Union[Tuple[torch.Tensor, ...], torch.Tensor]], Union[None, torch.Tensor]]*) → torch.utils.hooks.RemovableHandle

Registers a backward hook on the module.

The hook will be called every time the gradients with respect to a module are computed, i.e. the hook will execute if and only if the gradients with respect to module outputs are computed. The hook should have the following signature:

```
hook(module, grad_input, grad_output) -> tuple(Tensor) or None
```

The `grad_input` and `grad_output` are tuples that contain the gradients with respect to the inputs and outputs respectively. The hook should not modify its arguments, but it can optionally return a new gradient with respect to the input that will be used in place of `grad_input` in subsequent computations. `grad_input` will only correspond to the inputs given as positional arguments and all kwarg arguments are ignored. Entries in `grad_input` and `grad_output` will be `None` for all non-Tensor arguments.

For technical reasons, when this hook is applied to a Module, its forward function will receive a view of each Tensor passed to the Module. Similarly the caller will receive a view of each Tensor returned by the Module's forward function.

> **Warning:** Modifying inputs or outputs inplace is not allowed when using backward hooks and will raise an error.

Returns a handle that can be used to remove the added hook by calling `handle.remove()`

Return type `torch.utils.hooks.RemovableHandle`

register_load_state_dict_post_hook(*hook*)

Registers a post hook to be run after module's `load_state_dict` is called.

It should have the following signature:: hook(module, incompatible_keys) -> None

The `module` argument is the current module that this hook is registered on, and the `incompatible_keys` argument is a `NamedTuple` consisting of attributes `missing_keys` and `unexpected_keys`. `missing_keys` is a `list` of `str` containing the missing keys and `unexpected_keys` is a `list` of `str` containing the unexpected keys.

The given incompatible_keys can be modified inplace if needed.

Note that the checks performed when calling *load_state_dict()* with `strict=True` are affected by modifications the hook makes to `missing_keys` or `unexpected_keys`, as expected. Additions to either set of keys will result in an error being thrown when `strict=True`, and clearning out both missing and unexpected keys will avoid an error.

Returns a handle that can be used to remove the added hook by calling `handle.remove()`

Return type `torch.utils.hooks.RemovableHandle`

register_module(*name: str*, *module: Optional[torch.nn.modules.module.Module]*) → None
Alias for *add_module()*.

register_parameter(*name: str*, *param: Optional[torch.nn.parameter.Parameter]*) → None
Adds a parameter to the module.

The parameter can be accessed as an attribute using given name.

Parameters

- **name** (*str*) – name of the parameter. The parameter can be accessed from this module using the given name
- **param** (*Parameter or None*) – parameter to be added to the module. If `None`, then operations that run on parameters, such as *cuda*, are ignored. If `None`, the parameter is **not** included in the module's *state_dict*.

requires_grad_(*requires_grad: bool = True*) → torch.nn.modules.module.T
Change if autograd should record operations on parameters in this module.

This method sets the parameters' `requires_grad` attributes in-place.

This method is helpful for freezing part of the module for finetuning or training parts of a model individually (e.g., GAN training).

See locally-disable-grad-doc for a comparison between *.requires_grad_()* and several similar mechanisms that may be confused with it.

Parameters **requires_grad** (*bool*) – whether autograd should record operations on parameters in this module. Default: `True`.

Returns self

Return type Module

set_extra_state(*state: Any*)
This function is called from *load_state_dict()* to handle any extra state found within the *state_dict*. Implement this function and a corresponding *get_extra_state()* for your module if you need to store extra state within its *state_dict*.

Parameters **state** (*dict*) – Extra state from the *state_dict*

share_memory() → torch.nn.modules.module.T
See `torch.Tensor.share_memory_()`

state_dict(**args*, *destination=None*, *prefix=''*, *keep_vars=False*)

Returns a dictionary containing references to the whole state of the module.

Both parameters and persistent buffers (e.g. running averages) are included. Keys are corresponding parameter and buffer names. Parameters and buffers set to `None` are not included.

Note: The returned object is a shallow copy. It contains references to the module's parameters and buffers.

Warning: Currently `state_dict()` also accepts positional arguments for `destination`, `prefix` and `keep_vars` in order. However, this is being deprecated and keyword arguments will be enforced in future releases.

Warning: Please avoid the use of argument `destination` as it is not designed for end-users.

Parameters

- **destination** (*dict, optional*) – If provided, the state of module will be updated into the dict and the same object is returned. Otherwise, an `OrderedDict` will be created and returned. Default: `None`.
- **prefix** (*str, optional*) – a prefix added to parameter and buffer names to compose the keys in state_dict. Default: `''`.
- **keep_vars** (*bool, optional*) – by default the `Tensor` s returned in the state dict are detached from autograd. If it's set to `True`, detaching will not be performed. Default: `False`.

Returns a dictionary containing a whole state of the module

Return type dict

Example:

```
>>> # xdoctest: +SKIP("undefined vars")
>>> module.state_dict().keys()
['bias', 'weight']
```

to(**args*, ***kwargs*)

Moves and/or casts the parameters and buffers.

This can be called as

to(*device=None*, *dtype=None*, *non_blocking=False*)

to(*dtype*, *non_blocking=False*)

to(*tensor*, *non_blocking=False*)

to(*memory_format=torch.channels_last*)

Its signature is similar to `torch.Tensor.to()`, but only accepts floating point or complex `dtypes`. In addition, this method will only cast the floating point or complex parameters and buffers to `dtype` (if given). The integral parameters and buffers will be moved `device`, if that is given, but with dtypes unchanged. When `non_blocking` is set, it tries to convert/move asynchronously with respect to the host if possible, e.g., moving CPU Tensors with pinned memory to CUDA devices.

See below for examples.

Note: This method modifies the module in-place.

Parameters

- **device** (torch.device) – the desired device of the parameters and buffers in this module
- **dtype** (torch.dtype) – the desired floating point or complex dtype of the parameters and buffers in this module
- **tensor** (*torch.Tensor*) – Tensor whose dtype and device are the desired dtype and device for all parameters and buffers in this module
- **memory_format** (torch.memory_format) – the desired memory format for 4D parameters and buffers in this module (keyword only argument)

Returns self

Return type Module

Examples:

```
>>> # xdoctest: +IGNORE_WANT("non-deterministic")
>>> linear = nn.Linear(2, 2)
>>> linear.weight
Parameter containing:
tensor([[ 0.1913, -0.3420],
        [-0.5113, -0.2325]])
>>> linear.to(torch.double)
Linear(in_features=2, out_features=2, bias=True)
>>> linear.weight
Parameter containing:
tensor([[ 0.1913, -0.3420],
        [-0.5113, -0.2325]], dtype=torch.float64)
>>> # xdoctest: +REQUIRES(env:TORCH_DOCTEST_CUDA1)
>>> gpu1 = torch.device("cuda:1")
>>> linear.to(gpu1, dtype=torch.half, non_blocking=True)
Linear(in_features=2, out_features=2, bias=True)
>>> linear.weight
Parameter containing:
tensor([[ 0.1914, -0.3420],
        [-0.5112, -0.2324]], dtype=torch.float16, device='cuda:1')
>>> cpu = torch.device("cpu")
>>> linear.to(cpu)
Linear(in_features=2, out_features=2, bias=True)
>>> linear.weight
Parameter containing:
tensor([[ 0.1914, -0.3420],
        [-0.5112, -0.2324]], dtype=torch.float16)

>>> linear = nn.Linear(2, 2, bias=None).to(torch.cdouble)
>>> linear.weight
Parameter containing:
```

(continues on next page)

(continued from previous page)

```
tensor([[ 0.3741+0.j,  0.2382+0.j],
        [ 0.5593+0.j, -0.4443+0.j]], dtype=torch.complex128)
>>> linear(torch.ones(3, 2, dtype=torch.cdouble))
tensor([[0.6122+0.j, 0.1150+0.j],
        [0.6122+0.j, 0.1150+0.j],
        [0.6122+0.j, 0.1150+0.j]], dtype=torch.complex128)
```

to_empty(*, *device: Union[str, torch.device]*) → torch.nn.modules.module.T
Moves the parameters and buffers to the specified device without copying storage.

Parameters **device** (torch.device) – The desired device of the parameters and buffers in this module.

Returns self

Return type Module

train(*mode: bool = True*) → torch.nn.modules.module.T
Sets the module in training mode.

This has any effect only on certain modules. See documentations of particular modules for details of their behaviors in training/evaluation mode, if they are affected, e.g. Dropout, BatchNorm, etc.

Parameters **mode** (*bool*) – whether to set training mode (True) or evaluation mode (False). Default: True.

Returns self

Return type Module

training: bool

type(*dst_type: Union[torch.dtype, str]*) → torch.nn.modules.module.T
Casts all parameters and buffers to dst_type.

Note: This method modifies the module in-place.

Parameters **dst_type** (*type or string*) – the desired type

Returns self

Return type Module

xpu(*device: Optional[Union[int, torch.device]] = None*) → torch.nn.modules.module.T
Moves all model parameters and buffers to the XPU.

This also makes associated parameters and buffers different objects. So it should be called before constructing optimizer if the module will live on XPU while being optimized.

Note: This method modifies the module in-place.

Parameters **device** (*int, optional*) – if specified, all parameters will be copied to that device

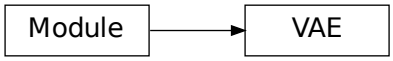
Returns self

Return type Module

zero_grad(*set_to_none: bool = False*) → None
Sets gradients of all model parameters to zero. See similar function under `torch.optim.Optimizer` for more context.

Parameters **set_to_none** (*bool*) – instead of setting to zero, set the grads to None. See `torch.optim.Optimizer.zero_grad()` for details.

class VAE(*n_features: int*, *n_units_embedding: int*, *n_units_conditional: int = 0*, *batch_size: int = 100*, *n_iter: int = 500*, *random_state: int = 0*, *lr: float = 0.0002*, *weight_decay: float = 0.001*, *decoder_n_layers_hidden: int = 2*, *decoder_n_units_hidden: int = 250*, *decoder_nonlin: str = 'leaky_relu'*, *decoder_nonlin_out: Optional[List[Tuple[str, int]]] = None*, *decoder_batch_norm: bool = False*, *decoder_dropout: float = 0*, *decoder_residual: bool = True*, *encoder_n_layers_hidden: int = 3*, *encoder_n_units_hidden: int = 300*, *encoder_nonlin: str = 'leaky_relu'*, *encoder_batch_norm: bool = False*, *encoder_dropout: float = 0.1*, *loss_strategy: str = 'standard'*, *loss_factor: int = 2*, *robust_divergence_beta: int = 2*, *dataloader_sampler: Optional[torch.utils.data.sampler.Sampler] = None*, *device: Any = device(type='cpu')*, *extra_loss_cbks: List[Callable] = []*, *clipping_value: int = 1*, *n_iter_min: int = 100*, *n_iter_print: int = 10*, *patience: int = 20*)
Bases: `torch.nn.modules.module.Module`



Basic VAE implementation.

Parameters

- **n_features** – int Number of features in the dataset
- **n_units_embedding** – int Number of units in the latent space
- **batch_size** – int Training batch size
- **n_iter** – int Number of training iterations
- **random_state** – int Random random_state
- **lr** – float Learning rate
- **weight_decay** – float: Optimizer weight decay
- **decoder_n_layers_hidden** – int Number of hidden layers in the decoder
- **decoder_n_units_hidden** – int Number of units in the hidden layer in the decoder
- **decoder_nonlin_out** – List List of activations layout, as generated by the tabular encoder
- **decoder_batch_norm** – bool Use batchnorm in the decoder
- **decoder_dropout** – float Use dropout in the decoder
- **decoder_residual** – bool Use residual connections in the decoder
- **encoder_n_layers_hidden** – int Number of hidden layers in the encoder
- **encoder_n_units_hidden** – int Number of units in the hidden layer in the encoder
- **encoder_batch_norm** – bool Use batchnorm in the encoder

- **encoder_dropout** – float Use dropout in the encoder
- **encoder_residual** – bool Use residual connections in the encoder
- **loss_strategy** – str - standard: classic VAE loss - robust_divergence: Algorithm 1 in "Robust Variational Autoencoder for Tabular Data with Divergence"
- **loss_factor** – int Parameter for the standard loss
- **robust_divergence_beta** – int Parameter for the robust_divergence loss
- **dataloader_sampler** – Custom sampler used by the dataloader, useful for conditional sampling.
- **device** – CPU/CUDA
- **extra_loss_cbks** – Custom loss callbacks. For example, for conditional loss.
- **clipping_value** – Gradients clipping value. Zero disables the feature
- **stopping** (*# early*) –
- **n_iter_print** – int Number of iterations after which to print updates and check the validation loss.
- **n_iter_min** – int Minimum number of iterations to go through before starting early stopping
- **patience** – int Max number of iterations without any improvement before early stopping is trigged.

T_destination
alias of TypeVar('T_destination', bound=Dict[str, Any])

add_module(*name: str*, *module: Optional[torch.nn.modules.module.Module]*) → None
Adds a child module to the current module.

The module can be accessed as an attribute using the given name.

Parameters

- **name** (*str*) – name of the child module. The child module can be accessed from this module using the given name
- **module** (*Module*) – child module to be added to the module.

apply(*fn: Callable[[torch.nn.modules.module.Module], None]*) → torch.nn.modules.module.T
Applies `fn` recursively to every submodule (as returned by `.children()`) as well as self. Typical use includes initializing the parameters of a model (see also nn-init-doc).

Parameters **fn** (Module -> None) – function to be applied to each submodule

Returns self

Return type Module

Example:

```
>>> @torch.no_grad()
>>> def init_weights(m):
>>>     print(m)
>>>     if type(m) == nn.Linear:
>>>         m.weight.fill_(1.0)
>>>         print(m.weight)
>>> net = nn.Sequential(nn.Linear(2, 2), nn.Linear(2, 2))
```

(continues on next page)

(continued from previous page)

```
>>> net.apply(init_weights)
Linear(in_features=2, out_features=2, bias=True)
Parameter containing:
tensor([[1., 1.],
        [1., 1.]], requires_grad=True)
Linear(in_features=2, out_features=2, bias=True)
Parameter containing:
tensor([[1., 1.],
        [1., 1.]], requires_grad=True)
Sequential(
  (0): Linear(in_features=2, out_features=2, bias=True)
  (1): Linear(in_features=2, out_features=2, bias=True)
)
```

bfloat16() → torch.nn.modules.module.T
Casts all floating point parameters and buffers to `bfloat16` datatype.

Note: This method modifies the module in-place.

Returns self

Return type Module

buffers(*recurse: bool = True*) → Iterator[torch.Tensor]
Returns an iterator over module buffers.

Parameters **recurse** (*bool*) – if True, then yields buffers of this module and all submodules. Otherwise, yields only buffers that are direct members of this module.

Yields *torch.Tensor* – module buffer

Example:

```
>>> # xdoctest: +SKIP("undefined vars")
>>> for buf in model.buffers():
>>>     print(type(buf), buf.size())
<class 'torch.Tensor'> (20L,)
<class 'torch.Tensor'> (20L, 1L, 5L, 5L)
```

children() → Iterator[torch.nn.modules.module.Module]
Returns an iterator over immediate children modules.

Yields *Module* – a child module

cpu() → torch.nn.modules.module.T
Moves all model parameters and buffers to the CPU.

Note: This method modifies the module in-place.

Returns self

Return type Module

cuda(*device: Optional[Union[int, torch.device]] = None*) → torch.nn.modules.module.T
Moves all model parameters and buffers to the GPU.

This also makes associated parameters and buffers different objects. So it should be called before constructing optimizer if the module will live on GPU while being optimized.

Note: This method modifies the module in-place.

Parameters **device** (*int, optional*) – if specified, all parameters will be copied to that device

Returns self

Return type Module

double() → torch.nn.modules.module.T
Casts all floating point parameters and buffers to `double` datatype.

Note: This method modifies the module in-place.

Returns self

Return type Module

dump_patches: bool = False

eval() → torch.nn.modules.module.T
Sets the module in evaluation mode.

This has any effect only on certain modules. See documentations of particular modules for details of their behaviors in training/evaluation mode, if they are affected, e.g. `Dropout`, `BatchNorm`, etc.

This is equivalent with `self.train(False)`.

See locally-disable-grad-doc for a comparison between *.eval()* and several similar mechanisms that may be confused with it.

Returns self

Return type Module

extra_repr() → str
Set the extra representation of the module

To print customized extra information, you should re-implement this method in your own modules. Both single-line and multi-line strings are acceptable.

fit(*X: numpy.ndarray*, *cond: Optional[numpy.ndarray] = None*) → Any

float() → torch.nn.modules.module.T
Casts all floating point parameters and buffers to `float` datatype.

Note: This method modifies the module in-place.

Returns self

Return type Module

forward(**input: Any*) → None
Defines the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

---

generate(*count: int*, *cond: Optional[numpy.ndarray] = None*) → numpy.ndarray

get_buffer(*target: str*) → torch.Tensor
Returns the buffer given by `target` if it exists, otherwise throws an error.

See the docstring for `get_submodule` for a more detailed explanation of this method's functionality as well as how to correctly specify `target`.

Parameters **target** – The fully-qualified string name of the buffer to look for. (See `get_submodule` for how to specify a fully-qualified string.)

Returns The buffer referenced by `target`

Return type torch.Tensor

Raises **AttributeError** – If the target string references an invalid path or resolves to something that is not a buffer

get_extra_state() → Any
Returns any extra state to include in the module's state_dict. Implement this and a corresponding *set_extra_state()* for your module if you need to store extra state. This function is called when building the module's *state_dict()*.

Note that extra state should be pickleable to ensure working serialization of the state_dict. We only provide provide backwards compatibility guarantees for serializing Tensors; other objects may break backwards compatibility if their serialized pickled form changes.

Returns Any extra state to store in the module's state_dict

Return type object

get_parameter(*target: str*) → torch.nn.parameter.Parameter
Returns the parameter given by `target` if it exists, otherwise throws an error.

See the docstring for `get_submodule` for a more detailed explanation of this method's functionality as well as how to correctly specify `target`.

Parameters **target** – The fully-qualified string name of the Parameter to look for. (See `get_submodule` for how to specify a fully-qualified string.)

Returns The Parameter referenced by `target`

Return type torch.nn.Parameter

Raises **AttributeError** – If the target string references an invalid path or resolves to something that is not an `nn.Parameter`

get_submodule(*target: str*) → torch.nn.modules.module.Module
Returns the submodule given by `target` if it exists, otherwise throws an error.

For example, let's say you have an `nn.Module` `A` that looks like this:

```
A(
    (net_b): Module(
        (net_c): Module(
            (conv): Conv2d(16, 33, kernel_size=(3, 3), stride=(2, 2))
        )
        (linear): Linear(in_features=100, out_features=200, bias=True)
    )
)
```

(The diagram shows an `nn.Module` `A`. `A` has a nested submodule `net_b`, which itself has two submodules `net_c` and `linear`. `net_c` then has a submodule `conv`.)

To check whether or not we have the `linear` submodule, we would call `get_submodule("net_b.linear")`. To check whether we have the `conv` submodule, we would call `get_submodule("net_b.net_c.conv")`.

The runtime of `get_submodule` is bounded by the degree of module nesting in `target`. A query against `named_modules` achieves the same result, but it is O(N) in the number of transitive modules. So, for a simple check to see if some submodule exists, `get_submodule` should always be used.

Parameters `target` – The fully-qualified string name of the submodule to look for. (See above example for how to specify a fully-qualified string.)

Returns The submodule referenced by `target`

Return type torch.nn.Module

Raises `AttributeError` – If the target string references an invalid path or resolves to something that is not an `nn.Module`

`half`() → torch.nn.modules.module.T

Casts all floating point parameters and buffers to `half` datatype.

Note: This method modifies the module in-place.

Returns self

Return type Module

`ipu`(*device: Optional[Union[int, torch.device]] = None*) → torch.nn.modules.module.T

Moves all model parameters and buffers to the IPU.

This also makes associated parameters and buffers different objects. So it should be called before constructing optimizer if the module will live on IPU while being optimized.

Note: This method modifies the module in-place.

Parameters `device` (*int, optional*) – if specified, all parameters will be copied to that device

Returns self

Return type Module

load_state_dict(*state_dict: Mapping[str, Any]*, *strict: bool = True*)

Copies parameters and buffers from *state_dict* into this module and its descendants. If `strict` is `True`, then the keys of *state_dict* must exactly match the keys returned by this module's `state_dict()` function.

Parameters

- **state_dict** (*dict*) – a dict containing parameters and persistent buffers.
- **strict** (*bool, optional*) – whether to strictly enforce that the keys in *state_dict* match the keys returned by this module's `state_dict()` function. Default: `True`

Returns

- **missing_keys** is a list of str containing the missing keys
- **unexpected_keys** is a list of str containing the unexpected keys

Return type `NamedTuple` with `missing_keys` and `unexpected_keys` fields

Note: If a parameter or buffer is registered as `None` and its corresponding key exists in *state_dict*, *load_state_dict()* will raise a `RuntimeError`.

---

modules() → Iterator[torch.nn.modules.module.Module]

Returns an iterator over all modules in the network.

Yields *Module* – a module in the network

Note: Duplicate modules are returned only once. In the following example, `l` will be returned only once.

Example:

```
>>> l = nn.Linear(2, 2)
>>> net = nn.Sequential(l, l)
>>> for idx, m in enumerate(net.modules()):
...     print(idx, '->', m)

0 -> Sequential(
  (0): Linear(in_features=2, out_features=2, bias=True)
  (1): Linear(in_features=2, out_features=2, bias=True)
)
1 -> Linear(in_features=2, out_features=2, bias=True)
```

named_buffers(*prefix: str = ''*, *recurse: bool = True*) → Iterator[Tuple[str, torch.Tensor]]

Returns an iterator over module buffers, yielding both the name of the buffer as well as the buffer itself.

Parameters

- **prefix** (*str*) – prefix to prepend to all buffer names.
- **recurse** (*bool*) – if True, then yields buffers of this module and all submodules. Otherwise, yields only buffers that are direct members of this module.

Yields *(str, torch.Tensor)* – Tuple containing the name and buffer

Example:

```
>>> # xdoctest: +SKIP("undefined vars")
>>> for name, buf in self.named_buffers():
>>>    if name in ['running_var']:
>>>        print(buf.size())
```

named_children() → Iterator[Tuple[str, torch.nn.modules.module.Module]]

Returns an iterator over immediate children modules, yielding both the name of the module as well as the module itself.

Yields *(str, Module)* – Tuple containing a name and child module

Example:

```
>>> # xdoctest: +SKIP("undefined vars")
>>> for name, module in model.named_children():
>>>     if name in ['conv4', 'conv5']:
>>>         print(module)
```

named_modules(*memo: Optional[Set[torch.nn.modules.module.Module]] = None*, *prefix: str = ''*, *remove_duplicate: bool = True*)

Returns an iterator over all modules in the network, yielding both the name of the module as well as the module itself.

Parameters

- **memo** – a memo to store the set of modules already added to the result
- **prefix** – a prefix that will be added to the name of the module
- **remove_duplicate** – whether to remove the duplicated module instances in the result or not

Yields *(str, Module)* – Tuple of name and module

Note: Duplicate modules are returned only once. In the following example, `l` will be returned only once.

Example:

```
>>> l = nn.Linear(2, 2)
>>> net = nn.Sequential(l, l)
>>> for idx, m in enumerate(net.named_modules()):
...     print(idx, '->', m)

0 -> ('', Sequential(
  (0): Linear(in_features=2, out_features=2, bias=True)
  (1): Linear(in_features=2, out_features=2, bias=True)
))
1 -> ('0', Linear(in_features=2, out_features=2, bias=True))
```

named_parameters(*prefix: str = ''*, *recurse: bool = True*) → Iterator[Tuple[str, torch.nn.parameter.Parameter]]

Returns an iterator over module parameters, yielding both the name of the parameter as well as the parameter itself.

Parameters

- **prefix** (*str*) – prefix to prepend to all parameter names.

- **recurse** (*bool*) – if True, then yields parameters of this module and all submodules. Otherwise, yields only parameters that are direct members of this module.

Yields *(str, Parameter)* – Tuple containing the name and parameter

Example:

```
>>> # xdoctest: +SKIP("undefined vars")
>>> for name, param in self.named_parameters():
>>>    if name in ['bias']:
>>>        print(param.size())
```

parameters(*recurse: bool = True*) → Iterator[torch.nn.parameter.Parameter]

Returns an iterator over module parameters.

This is typically passed to an optimizer.

Parameters **recurse** (*bool*) – if True, then yields parameters of this module and all submodules. Otherwise, yields only parameters that are direct members of this module.

Yields *Parameter* – module parameter

Example:

```
>>> # xdoctest: +SKIP("undefined vars")
>>> for param in model.parameters():
>>>     print(type(param), param.size())
<class 'torch.Tensor'> (20L,)
<class 'torch.Tensor'> (20L, 1L, 5L, 5L)
```

register_backward_hook(*hook: Callable[[torch.nn.modules.module.Module, Union[Tuple[torch.Tensor, ...], torch.Tensor], Union[Tuple[torch.Tensor, ...], torch.Tensor]], Union[None, torch.Tensor]]*) → torch.utils.hooks.RemovableHandle

Registers a backward hook on the module.

This function is deprecated in favor of `register_full_backward_hook()` and the behavior of this function will change in future versions.

Returns a handle that can be used to remove the added hook by calling `handle.remove()`

Return type `torch.utils.hooks.RemovableHandle`

register_buffer(*name: str*, *tensor: Optional[torch.Tensor]*, *persistent: bool = True*) → None

Adds a buffer to the module.

This is typically used to register a buffer that should not to be considered a model parameter. For example, BatchNorm's `running_mean` is not a parameter, but is part of the module's state. Buffers, by default, are persistent and will be saved alongside parameters. This behavior can be changed by setting `persistent` to `False`. The only difference between a persistent buffer and a non-persistent buffer is that the latter will not be a part of this module's *state_dict*.

Buffers can be accessed as attributes using given names.

Parameters

- **name** (*str*) – name of the buffer. The buffer can be accessed from this module using the given name
- **tensor** (*Tensor or None*) – buffer to be registered. If `None`, then operations that run on buffers, such as *cuda*, are ignored. If `None`, the buffer is **not** included in the module's *state_dict*.

- **persistent** (*bool*) – whether the buffer is part of this module's *state_dict*.

Example:

```
>>> # xdoctest: +SKIP("undefined vars")
>>> self.register_buffer('running_mean', torch.zeros(num_features))
```

register_forward_hook(*hook: Callable[[...], None]*) → torch.utils.hooks.RemovableHandle

Registers a forward hook on the module.

The hook will be called every time after *forward()* has computed an output. It should have the following signature:

```
hook(module, input, output) -> None or modified output
```

The input contains only the positional arguments given to the module. Keyword arguments won't be passed to the hooks and only to the `forward`. The hook can modify the output. It can modify the input inplace but it will not have effect on forward since this is called after *forward()* is called.

Returns a handle that can be used to remove the added hook by calling `handle.remove()`

Return type `torch.utils.hooks.RemovableHandle`

register_forward_pre_hook(*hook: Callable[[...], None]*) → torch.utils.hooks.RemovableHandle

Registers a forward pre-hook on the module.

The hook will be called every time before *forward()* is invoked. It should have the following signature:

```
hook(module, input) -> None or modified input
```

The input contains only the positional arguments given to the module. Keyword arguments won't be passed to the hooks and only to the `forward`. The hook can modify the input. User can either return a tuple or a single modified value in the hook. We will wrap the value into a tuple if a single value is returned(unless that value is already a tuple).

Returns a handle that can be used to remove the added hook by calling `handle.remove()`

Return type `torch.utils.hooks.RemovableHandle`

register_full_backward_hook(*hook: Callable[[torch.nn.modules.module.Module, Union[Tuple[torch.Tensor, ...], torch.Tensor], Union[Tuple[torch.Tensor, ...], torch.Tensor]], Union[None, torch.Tensor]]*) → torch.utils.hooks.RemovableHandle

Registers a backward hook on the module.

The hook will be called every time the gradients with respect to a module are computed, i.e. the hook will execute if and only if the gradients with respect to module outputs are computed. The hook should have the following signature:

```
hook(module, grad_input, grad_output) -> tuple(Tensor) or None
```

The `grad_input` and `grad_output` are tuples that contain the gradients with respect to the inputs and outputs respectively. The hook should not modify its arguments, but it can optionally return a new gradient with respect to the input that will be used in place of `grad_input` in subsequent computations. `grad_input` will only correspond to the inputs given as positional arguments and all kwarg arguments are ignored. Entries in `grad_input` and `grad_output` will be `None` for all non-Tensor arguments.

For technical reasons, when this hook is applied to a Module, its forward function will receive a view of each Tensor passed to the Module. Similarly the caller will receive a view of each Tensor returned by the Module's forward function.

> **Warning:** Modifying inputs or outputs inplace is not allowed when using backward hooks and will raise an error.

Returns a handle that can be used to remove the added hook by calling `handle.remove()`

Return type `torch.utils.hooks.RemovableHandle`

register_load_state_dict_post_hook(*hook*)

Registers a post hook to be run after module's `load_state_dict` is called.

It should have the following signature:: hook(module, incompatible_keys) -> None

The `module` argument is the current module that this hook is registered on, and the `incompatible_keys` argument is a `NamedTuple` consisting of attributes `missing_keys` and `unexpected_keys`. `missing_keys` is a `list` of `str` containing the missing keys and `unexpected_keys` is a `list` of `str` containing the unexpected keys.

The given incompatible_keys can be modified inplace if needed.

Note that the checks performed when calling *load_state_dict()* with `strict=True` are affected by modifications the hook makes to `missing_keys` or `unexpected_keys`, as expected. Additions to either set of keys will result in an error being thrown when `strict=True`, and clearning out both missing and unexpected keys will avoid an error.

Returns a handle that can be used to remove the added hook by calling `handle.remove()`

Return type `torch.utils.hooks.RemovableHandle`

register_module(*name: str*, *module: Optional[torch.nn.modules.module.Module]*) → None

Alias for *add_module()*.

register_parameter(*name: str*, *param: Optional[torch.nn.parameter.Parameter]*) → None

Adds a parameter to the module.

The parameter can be accessed as an attribute using given name.

Parameters

- **name** (*str*) – name of the parameter. The parameter can be accessed from this module using the given name
- **param** (*Parameter or None*) – parameter to be added to the module. If `None`, then operations that run on parameters, such as *cuda*, are ignored. If `None`, the parameter is **not** included in the module's *state_dict*.

requires_grad_(*requires_grad: bool = True*) → torch.nn.modules.module.T

Change if autograd should record operations on parameters in this module.

This method sets the parameters' `requires_grad` attributes in-place.

This method is helpful for freezing part of the module for finetuning or training parts of a model individually (e.g., GAN training).

See locally-disable-grad-doc for a comparison between *.requires_grad_()* and several similar mechanisms that may be confused with it.

Parameters **requires_grad** (*bool*) – whether autograd should record operations on parameters in this module. Default: `True`.

Returns self

Return type Module

set_extra_state(*state: Any*)
This function is called from *load_state_dict()* to handle any extra state found within the *state_dict*. Implement this function and a corresponding *get_extra_state()* for your module if you need to store extra state within its *state_dict*.

Parameters **state** (*dict*) – Extra state from the *state_dict*

share_memory() → torch.nn.modules.module.T
See `torch.Tensor.share_memory_()`

state_dict(**args*, *destination=None*, *prefix=''*, *keep_vars=False*)
Returns a dictionary containing references to the whole state of the module.

Both parameters and persistent buffers (e.g. running averages) are included. Keys are corresponding parameter and buffer names. Parameters and buffers set to `None` are not included.

Note: The returned object is a shallow copy. It contains references to the module's parameters and buffers.

Warning: Currently `state_dict()` also accepts positional arguments for `destination`, `prefix` and `keep_vars` in order. However, this is being deprecated and keyword arguments will be enforced in future releases.

Warning: Please avoid the use of argument `destination` as it is not designed for end-users.

Parameters

- **destination** (*dict, optional*) – If provided, the state of module will be updated into the dict and the same object is returned. Otherwise, an `OrderedDict` will be created and returned. Default: `None`.
- **prefix** (*str, optional*) – a prefix added to parameter and buffer names to compose the keys in state_dict. Default: `''`.
- **keep_vars** (*bool, optional*) – by default the `Tensor` s returned in the state dict are detached from autograd. If it's set to `True`, detaching will not be performed. Default: `False`.

Returns a dictionary containing a whole state of the module

Return type dict

Example:

```
>>> # xdoctest: +SKIP("undefined vars")
>>> module.state_dict().keys()
['bias', 'weight']
```

to(**args*, ***kwargs*)
Moves and/or casts the parameters and buffers.

This can be called as

to(*device=None*, *dtype=None*, *non_blocking=False*)

to(*dtype*, *non_blocking=False*)

to(*tensor*, *non_blocking=False*)

to(*memory_format=torch.channels_last*)

Its signature is similar to `torch.Tensor.to()`, but only accepts floating point or complex `dtype`s. In addition, this method will only cast the floating point or complex parameters and buffers to `dtype` (if given). The integral parameters and buffers will be moved `device`, if that is given, but with dtypes unchanged. When `non_blocking` is set, it tries to convert/move asynchronously with respect to the host if possible, e.g., moving CPU Tensors with pinned memory to CUDA devices.

See below for examples.

Note: This method modifies the module in-place.

Parameters

- **device** (`torch.device`) – the desired device of the parameters and buffers in this module
- **dtype** (`torch.dtype`) – the desired floating point or complex dtype of the parameters and buffers in this module
- **tensor** (*torch.Tensor*) – Tensor whose dtype and device are the desired dtype and device for all parameters and buffers in this module
- **memory_format** (`torch.memory_format`) – the desired memory format for 4D parameters and buffers in this module (keyword only argument)

Returns self

Return type Module

Examples:

```
>>> # xdoctest: +IGNORE_WANT("non-deterministic")
>>> linear = nn.Linear(2, 2)
>>> linear.weight
Parameter containing:
tensor([[ 0.1913, -0.3420],
        [-0.5113, -0.2325]])
>>> linear.to(torch.double)
Linear(in_features=2, out_features=2, bias=True)
>>> linear.weight
Parameter containing:
tensor([[ 0.1913, -0.3420],
        [-0.5113, -0.2325]], dtype=torch.float64)
>>> # xdoctest: +REQUIRES(env:TORCH_DOCTEST_CUDA1)
>>> gpu1 = torch.device("cuda:1")
>>> linear.to(gpu1, dtype=torch.half, non_blocking=True)
Linear(in_features=2, out_features=2, bias=True)
>>> linear.weight
Parameter containing:
tensor([[ 0.1914, -0.3420],
        [-0.5112, -0.2324]], dtype=torch.float16, device='cuda:1')
>>> cpu = torch.device("cpu")
>>> linear.to(cpu)
```

(continues on next page)

(continued from previous page)

```
Linear(in_features=2, out_features=2, bias=True)
>>> linear.weight
Parameter containing:
tensor([[ 0.1914, -0.3420],
        [-0.5112, -0.2324]], dtype=torch.float16)

>>> linear = nn.Linear(2, 2, bias=None).to(torch.cdouble)
>>> linear.weight
Parameter containing:
tensor([[ 0.3741+0.j,  0.2382+0.j],
        [ 0.5593+0.j, -0.4443+0.j]], dtype=torch.complex128)
>>> linear(torch.ones(3, 2, dtype=torch.cdouble))
tensor([[0.6122+0.j, 0.1150+0.j],
        [0.6122+0.j, 0.1150+0.j],
        [0.6122+0.j, 0.1150+0.j]], dtype=torch.complex128)
```

to_empty(*, *device: Union[str, torch.device]*) → torch.nn.modules.module.T
Moves the parameters and buffers to the specified device without copying storage.

Parameters **device** (`torch.device`) – The desired device of the parameters and buffers in this module.

Returns self

Return type Module

train(*mode: bool = True*) → torch.nn.modules.module.T
Sets the module in training mode.

This has any effect only on certain modules. See documentations of particular modules for details of their behaviors in training/evaluation mode, if they are affected, e.g. `Dropout`, `BatchNorm`, etc.

Parameters **mode** (*bool*) – whether to set training mode (`True`) or evaluation mode (`False`). Default: `True`.

Returns self

Return type Module

training: bool

type(*dst_type: Union[torch.dtype, str]*) → torch.nn.modules.module.T
Casts all parameters and buffers to `dst_type`.

Note: This method modifies the module in-place.

Parameters **dst_type** (*type or string*) – the desired type

Returns self

Return type Module

xpu(*device: Optional[Union[int, torch.device]] = None*) → torch.nn.modules.module.T
Moves all model parameters and buffers to the XPU.

This also makes associated parameters and buffers different objects. So it should be called before constructing optimizer if the module will live on XPU while being optimized.

Note: This method modifies the module in-place.

Parameters **device** (*int, optional*) – if specified, all parameters will be copied to that device

Returns self

Return type Module

zero_grad(*set_to_none: bool = False*) → None
Sets gradients of all model parameters to zero. See similar function under `torch.optim.Optimizer` for more context.

Parameters **set_to_none** (*bool*) – instead of setting to zero, set the grads to None. See `torch.optim.Optimizer.zero_grad()` for details.

synthcity.plugins.core.models.transformer module

class TransformerModel(*n_units_in: int*, *n_units_hidden: int = 64*, *n_head: int = 1*, *d_ffn: int = 128*, *dropout: float = 0.1*, *activation: str = 'relu'*, *n_layers_hidden: int = 1*, *device: Any = device(type='cpu')*)
Bases: `torch.nn.modules.module.Module`

T_destination
alias of TypeVar('T_destination', bound=`Dict[str, Any]`)

add_module(*name: str*, *module: Optional[torch.nn.modules.module.Module]*) → None
Adds a child module to the current module.

The module can be accessed as an attribute using the given name.

Parameters

- **name** (*str*) – name of the child module. The child module can be accessed from this module using the given name
- **module** (*Module*) – child module to be added to the module.

apply(*fn: Callable[[torch.nn.modules.module.Module], None]*) → torch.nn.modules.module.T
Applies `fn` recursively to every submodule (as returned by `.children()`) as well as self. Typical use includes initializing the parameters of a model (see also nn-init-doc).

Parameters **fn** (`Module` -> None) – function to be applied to each submodule

Returns self

Return type Module

Example:

```
>>> @torch.no_grad()
>>> def init_weights(m):
>>>     print(m)
>>>     if type(m) == nn.Linear:
>>>         m.weight.fill_(1.0)
>>>         print(m.weight)
>>> net = nn.Sequential(nn.Linear(2, 2), nn.Linear(2, 2))
>>> net.apply(init_weights)
```

(continues on next page)

(continued from previous page)

```
Linear(in_features=2, out_features=2, bias=True)
Parameter containing:
tensor([[1., 1.],
        [1., 1.]], requires_grad=True)
Linear(in_features=2, out_features=2, bias=True)
Parameter containing:
tensor([[1., 1.],
        [1., 1.]], requires_grad=True)
Sequential(
  (0): Linear(in_features=2, out_features=2, bias=True)
  (1): Linear(in_features=2, out_features=2, bias=True)
)
```

bfloat16() → torch.nn.modules.module.T
Casts all floating point parameters and buffers to `bfloat16` datatype.

Note: This method modifies the module in-place.

Returns self

Return type Module

buffers(*recurse: bool = True*) → Iterator[torch.Tensor]
Returns an iterator over module buffers.

Parameters **recurse** (*bool*) – if True, then yields buffers of this module and all submodules. Otherwise, yields only buffers that are direct members of this module.

Yields *torch.Tensor* – module buffer

Example:

```
>>> # xdoctest: +SKIP("undefined vars")
>>> for buf in model.buffers():
>>>     print(type(buf), buf.size())
<class 'torch.Tensor'> (20L,)
<class 'torch.Tensor'> (20L, 1L, 5L, 5L)
```

children() → Iterator[torch.nn.modules.module.Module]
Returns an iterator over immediate children modules.

Yields *Module* – a child module

cpu() → torch.nn.modules.module.T
Moves all model parameters and buffers to the CPU.

Note: This method modifies the module in-place.

Returns self

Return type Module

cuda(*device: Optional[Union[int, torch.device]] = None*) → torch.nn.modules.module.T
Moves all model parameters and buffers to the GPU.

This also makes associated parameters and buffers different objects. So it should be called before constructing optimizer if the module will live on GPU while being optimized.

Note: This method modifies the module in-place.

Parameters **device** (*int, optional*) – if specified, all parameters will be copied to that device

Returns self

Return type Module

double() → torch.nn.modules.module.T
Casts all floating point parameters and buffers to `double` datatype.

Note: This method modifies the module in-place.

Returns self

Return type Module

dump_patches: bool = False

eval() → torch.nn.modules.module.T
Sets the module in evaluation mode.

This has any effect only on certain modules. See documentations of particular modules for details of their behaviors in training/evaluation mode, if they are affected, e.g. `Dropout`, `BatchNorm`, etc.

This is equivalent with `self.train(False)`.

See locally-disable-grad-doc for a comparison between *.eval()* and several similar mechanisms that may be confused with it.

Returns self

Return type Module

extra_repr() → str
Set the extra representation of the module

To print customized extra information, you should re-implement this method in your own modules. Both single-line and multi-line strings are acceptable.

float() → torch.nn.modules.module.T
Casts all floating point parameters and buffers to `float` datatype.

Note: This method modifies the module in-place.

Returns self

Return type Module

forward(*x: torch.Tensor*) → torch.Tensor
Defines the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

---

get_buffer(*target: str*) → torch.Tensor
Returns the buffer given by `target` if it exists, otherwise throws an error.

See the docstring for `get_submodule` for a more detailed explanation of this method's functionality as well as how to correctly specify `target`.

Parameters **target** – The fully-qualified string name of the buffer to look for. (See `get_submodule` for how to specify a fully-qualified string.)

Returns The buffer referenced by `target`

Return type torch.Tensor

Raises **AttributeError** – If the target string references an invalid path or resolves to something that is not a buffer

get_extra_state() → Any
Returns any extra state to include in the module's state_dict. Implement this and a corresponding *set_extra_state()* for your module if you need to store extra state. This function is called when building the module's *state_dict()*.

Note that extra state should be pickleable to ensure working serialization of the state_dict. We only provide provide backwards compatibility guarantees for serializing Tensors; other objects may break backwards compatibility if their serialized pickled form changes.

Returns Any extra state to store in the module's state_dict

Return type object

get_parameter(*target: str*) → torch.nn.parameter.Parameter
Returns the parameter given by `target` if it exists, otherwise throws an error.

See the docstring for `get_submodule` for a more detailed explanation of this method's functionality as well as how to correctly specify `target`.

Parameters **target** – The fully-qualified string name of the Parameter to look for. (See `get_submodule` for how to specify a fully-qualified string.)

Returns The Parameter referenced by `target`

Return type torch.nn.Parameter

Raises **AttributeError** – If the target string references an invalid path or resolves to something that is not an `nn.Parameter`

get_submodule(*target: str*) → torch.nn.modules.module.Module
Returns the submodule given by `target` if it exists, otherwise throws an error.

For example, let's say you have an `nn.Module` `A` that looks like this:

```
A(
    (net_b): Module(
```

(continues on next page)

(continued from previous page)

```
        (net_c): Module(
            (conv): Conv2d(16, 33, kernel_size=(3, 3), stride=(2, 2))
        )
        (linear): Linear(in_features=100, out_features=200, bias=True)
    )
)
```

(The diagram shows an `nn.Module` `A`. `A` has a nested submodule `net_b`, which itself has two submodules `net_c` and `linear`. `net_c` then has a submodule `conv`.)

To check whether or not we have the `linear` submodule, we would call `get_submodule("net_b.linear")`. To check whether we have the `conv` submodule, we would call `get_submodule("net_b.net_c.conv")`.

The runtime of `get_submodule` is bounded by the degree of module nesting in `target`. A query against `named_modules` achieves the same result, but it is O(N) in the number of transitive modules. So, for a simple check to see if some submodule exists, `get_submodule` should always be used.

Parameters `target` – The fully-qualified string name of the submodule to look for. (See above example for how to specify a fully-qualified string.)

Returns The submodule referenced by `target`

Return type torch.nn.Module

Raises `AttributeError` – If the target string references an invalid path or resolves to something that is not an `nn.Module`

`half`() → torch.nn.modules.module.T

Casts all floating point parameters and buffers to `half` datatype.

Note: This method modifies the module in-place.

Returns self

Return type Module

`ipu`(*device: Optional[Union[int, torch.device]] = None*) → torch.nn.modules.module.T

Moves all model parameters and buffers to the IPU.

This also makes associated parameters and buffers different objects. So it should be called before constructing optimizer if the module will live on IPU while being optimized.

Note: This method modifies the module in-place.

Parameters `device` (*int, optional*) – if specified, all parameters will be copied to that device

Returns self

Return type Module

load_state_dict(*state_dict: Mapping[str, Any]*, *strict: bool = True*)

Copies parameters and buffers from *state_dict* into this module and its descendants. If `strict` is `True`, then the keys of *state_dict* must exactly match the keys returned by this module's `state_dict()` function.

Parameters

- **state_dict** (*dict*) – a dict containing parameters and persistent buffers.
- **strict** (*bool, optional*) – whether to strictly enforce that the keys in *state_dict* match the keys returned by this module's `state_dict()` function. Default: `True`

Returns

- **missing_keys** is a list of str containing the missing keys
- **unexpected_keys** is a list of str containing the unexpected keys

Return type `NamedTuple` with `missing_keys` and `unexpected_keys` fields

Note: If a parameter or buffer is registered as `None` and its corresponding key exists in *state_dict*, *load_state_dict()* will raise a `RuntimeError`.

---

modules() → Iterator[torch.nn.modules.module.Module]

Returns an iterator over all modules in the network.

Yields *Module* – a module in the network

Note: Duplicate modules are returned only once. In the following example, `l` will be returned only once.

Example:

```
>>> l = nn.Linear(2, 2)
>>> net = nn.Sequential(l, l)
>>> for idx, m in enumerate(net.modules()):
...     print(idx, '->', m)

0 -> Sequential(
  (0): Linear(in_features=2, out_features=2, bias=True)
  (1): Linear(in_features=2, out_features=2, bias=True)
)
1 -> Linear(in_features=2, out_features=2, bias=True)
```

named_buffers(*prefix: str = ''*, *recurse: bool = True*) → Iterator[Tuple[str, torch.Tensor]]

Returns an iterator over module buffers, yielding both the name of the buffer as well as the buffer itself.

Parameters

- **prefix** (*str*) – prefix to prepend to all buffer names.
- **recurse** (*bool*) – if True, then yields buffers of this module and all submodules. Otherwise, yields only buffers that are direct members of this module.

Yields *(str, torch.Tensor)* – Tuple containing the name and buffer

Example:

```
>>> # xdoctest: +SKIP("undefined vars")
>>> for name, buf in self.named_buffers():
>>>    if name in ['running_var']:
>>>        print(buf.size())
```

named_children() → Iterator[Tuple[str, torch.nn.modules.module.Module]]

Returns an iterator over immediate children modules, yielding both the name of the module as well as the module itself.

Yields *(str, Module)* – Tuple containing a name and child module

Example:

```
>>> # xdoctest: +SKIP("undefined vars")
>>> for name, module in model.named_children():
>>>     if name in ['conv4', 'conv5']:
>>>         print(module)
```

named_modules(*memo: Optional[Set[torch.nn.modules.module.Module]] = None*, *prefix: str = ''*, *remove_duplicate: bool = True*)

Returns an iterator over all modules in the network, yielding both the name of the module as well as the module itself.

Parameters

- **memo** – a memo to store the set of modules already added to the result
- **prefix** – a prefix that will be added to the name of the module
- **remove_duplicate** – whether to remove the duplicated module instances in the result or not

Yields *(str, Module)* – Tuple of name and module

Note: Duplicate modules are returned only once. In the following example, `l` will be returned only once.

Example:

```
>>> l = nn.Linear(2, 2)
>>> net = nn.Sequential(l, l)
>>> for idx, m in enumerate(net.named_modules()):
...     print(idx, '->', m)

0 -> ('', Sequential(
  (0): Linear(in_features=2, out_features=2, bias=True)
  (1): Linear(in_features=2, out_features=2, bias=True)
))
1 -> ('0', Linear(in_features=2, out_features=2, bias=True))
```

named_parameters(*prefix: str = ''*, *recurse: bool = True*) → Iterator[Tuple[str, torch.nn.parameter.Parameter]]

Returns an iterator over module parameters, yielding both the name of the parameter as well as the parameter itself.

Parameters

- **prefix** (*str*) – prefix to prepend to all parameter names.

- **recurse** (*bool*) – if True, then yields parameters of this module and all submodules. Otherwise, yields only parameters that are direct members of this module.

Yields *(str, Parameter)* – Tuple containing the name and parameter

Example:

```
>>> # xdoctest: +SKIP("undefined vars")
>>> for name, param in self.named_parameters():
>>>    if name in ['bias']:
>>>        print(param.size())
```

parameters(*recurse: bool = True*) → Iterator[torch.nn.parameter.Parameter]

Returns an iterator over module parameters.

This is typically passed to an optimizer.

Parameters **recurse** (*bool*) – if True, then yields parameters of this module and all submodules. Otherwise, yields only parameters that are direct members of this module.

Yields *Parameter* – module parameter

Example:

```
>>> # xdoctest: +SKIP("undefined vars")
>>> for param in model.parameters():
>>>     print(type(param), param.size())
<class 'torch.Tensor'> (20L,)
<class 'torch.Tensor'> (20L, 1L, 5L, 5L)
```

register_backward_hook(*hook: Callable[[torch.nn.modules.module.Module, Union[Tuple[torch.Tensor, ...], torch.Tensor], Union[Tuple[torch.Tensor, ...], torch.Tensor]], Union[None, torch.Tensor]]*) → torch.utils.hooks.RemovableHandle

Registers a backward hook on the module.

This function is deprecated in favor of `register_full_backward_hook()` and the behavior of this function will change in future versions.

Returns a handle that can be used to remove the added hook by calling `handle.remove()`

Return type `torch.utils.hooks.RemovableHandle`

register_buffer(*name: str*, *tensor: Optional[torch.Tensor]*, *persistent: bool = True*) → None

Adds a buffer to the module.

This is typically used to register a buffer that should not to be considered a model parameter. For example, BatchNorm's `running_mean` is not a parameter, but is part of the module's state. Buffers, by default, are persistent and will be saved alongside parameters. This behavior can be changed by setting `persistent` to `False`. The only difference between a persistent buffer and a non-persistent buffer is that the latter will not be a part of this module's *state_dict*.

Buffers can be accessed as attributes using given names.

Parameters

- **name** (*str*) – name of the buffer. The buffer can be accessed from this module using the given name
- **tensor** (*Tensor or None*) – buffer to be registered. If `None`, then operations that run on buffers, such as *cuda*, are ignored. If `None`, the buffer is **not** included in the module's *state_dict*.

- **persistent** (*bool*) – whether the buffer is part of this module's *state_dict*.

Example:

```
>>> # xdoctest: +SKIP("undefined vars")
>>> self.register_buffer('running_mean', torch.zeros(num_features))
```

register_forward_hook(*hook: Callable[[...], None]*) → torch.utils.hooks.RemovableHandle

Registers a forward hook on the module.

The hook will be called every time after *forward()* has computed an output. It should have the following signature:

```
hook(module, input, output) -> None or modified output
```

The input contains only the positional arguments given to the module. Keyword arguments won't be passed to the hooks and only to the `forward`. The hook can modify the output. It can modify the input inplace but it will not have effect on forward since this is called after *forward()* is called.

Returns a handle that can be used to remove the added hook by calling `handle.remove()`

Return type `torch.utils.hooks.RemovableHandle`

register_forward_pre_hook(*hook: Callable[[...], None]*) → torch.utils.hooks.RemovableHandle

Registers a forward pre-hook on the module.

The hook will be called every time before *forward()* is invoked. It should have the following signature:

```
hook(module, input) -> None or modified input
```

The input contains only the positional arguments given to the module. Keyword arguments won't be passed to the hooks and only to the `forward`. The hook can modify the input. User can either return a tuple or a single modified value in the hook. We will wrap the value into a tuple if a single value is returned(unless that value is already a tuple).

Returns a handle that can be used to remove the added hook by calling `handle.remove()`

Return type `torch.utils.hooks.RemovableHandle`

register_full_backward_hook(*hook: Callable[[torch.nn.modules.module.Module, Union[Tuple[torch.Tensor, ...], torch.Tensor], Union[Tuple[torch.Tensor, ...], torch.Tensor]], Union[None, torch.Tensor]]*) → torch.utils.hooks.RemovableHandle

Registers a backward hook on the module.

The hook will be called every time the gradients with respect to a module are computed, i.e. the hook will execute if and only if the gradients with respect to module outputs are computed. The hook should have the following signature:

```
hook(module, grad_input, grad_output) -> tuple(Tensor) or None
```

The `grad_input` and `grad_output` are tuples that contain the gradients with respect to the inputs and outputs respectively. The hook should not modify its arguments, but it can optionally return a new gradient with respect to the input that will be used in place of `grad_input` in subsequent computations. `grad_input` will only correspond to the inputs given as positional arguments and all kwarg arguments are ignored. Entries in `grad_input` and `grad_output` will be `None` for all non-Tensor arguments.

For technical reasons, when this hook is applied to a Module, its forward function will receive a view of each Tensor passed to the Module. Similarly the caller will receive a view of each Tensor returned by the Module's forward function.

> **Warning:** Modifying inputs or outputs inplace is not allowed when using backward hooks and will raise an error.

Returns a handle that can be used to remove the added hook by calling `handle.remove()`

Return type `torch.utils.hooks.RemovableHandle`

register_load_state_dict_post_hook(*hook*)

Registers a post hook to be run after module's `load_state_dict` is called.

It should have the following signature:: hook(module, incompatible_keys) -> None

The `module` argument is the current module that this hook is registered on, and the `incompatible_keys` argument is a `NamedTuple` consisting of attributes `missing_keys` and `unexpected_keys`. `missing_keys` is a `list` of `str` containing the missing keys and `unexpected_keys` is a `list` of `str` containing the unexpected keys.

The given incompatible_keys can be modified inplace if needed.

Note that the checks performed when calling *load_state_dict()* with `strict=True` are affected by modifications the hook makes to `missing_keys` or `unexpected_keys`, as expected. Additions to either set of keys will result in an error being thrown when `strict=True`, and clearning out both missing and unexpected keys will avoid an error.

Returns a handle that can be used to remove the added hook by calling `handle.remove()`

Return type `torch.utils.hooks.RemovableHandle`

register_module(*name: str*, *module: Optional[torch.nn.modules.module.Module]*) → None

Alias for *add_module()*.

register_parameter(*name: str*, *param: Optional[torch.nn.parameter.Parameter]*) → None

Adds a parameter to the module.

The parameter can be accessed as an attribute using given name.

Parameters

- **name** (*str*) – name of the parameter. The parameter can be accessed from this module using the given name
- **param** (*Parameter or None*) – parameter to be added to the module. If `None`, then operations that run on parameters, such as *cuda*, are ignored. If `None`, the parameter is **not** included in the module's *state_dict*.

requires_grad_(*requires_grad: bool = True*) → torch.nn.modules.module.T

Change if autograd should record operations on parameters in this module.

This method sets the parameters' `requires_grad` attributes in-place.

This method is helpful for freezing part of the module for finetuning or training parts of a model individually (e.g., GAN training).

See locally-disable-grad-doc for a comparison between *.requires_grad_()* and several similar mechanisms that may be confused with it.

Parameters **requires_grad** (*bool*) – whether autograd should record operations on parameters in this module. Default: `True`.

Returns self

Return type Module

set_extra_state(*state: Any*)
This function is called from *load_state_dict()* to handle any extra state found within the *state_dict*. Implement this function and a corresponding *get_extra_state()* for your module if you need to store extra state within its *state_dict*.

Parameters **state** (*dict*) – Extra state from the *state_dict*

share_memory() → torch.nn.modules.module.T
See `torch.Tensor.share_memory_()`

state_dict(**args*, *destination=None*, *prefix=''*, *keep_vars=False*)
Returns a dictionary containing references to the whole state of the module.

Both parameters and persistent buffers (e.g. running averages) are included. Keys are corresponding parameter and buffer names. Parameters and buffers set to `None` are not included.

Note: The returned object is a shallow copy. It contains references to the module's parameters and buffers.

Warning: Currently `state_dict()` also accepts positional arguments for `destination`, `prefix` and `keep_vars` in order. However, this is being deprecated and keyword arguments will be enforced in future releases.

Warning: Please avoid the use of argument `destination` as it is not designed for end-users.

Parameters

- **destination** (*dict, optional*) – If provided, the state of module will be updated into the dict and the same object is returned. Otherwise, an `OrderedDict` will be created and returned. Default: `None`.
- **prefix** (*str, optional*) – a prefix added to parameter and buffer names to compose the keys in state_dict. Default: `''`.
- **keep_vars** (*bool, optional*) – by default the `Tensor` s returned in the state dict are detached from autograd. If it's set to `True`, detaching will not be performed. Default: `False`.

Returns a dictionary containing a whole state of the module

Return type dict

Example:

```
>>> # xdoctest: +SKIP("undefined vars")
>>> module.state_dict().keys()
['bias', 'weight']
```

to(**args*, ***kwargs*)
Moves and/or casts the parameters and buffers.

This can be called as

to(*device=None*, *dtype=None*, *non_blocking=False*)

to(*dtype*, *non_blocking=False*)

to(*tensor*, *non_blocking=False*)

to(*memory_format=torch.channels_last*)

Its signature is similar to `torch.Tensor.to()`, but only accepts floating point or complex `dtypes`. In addition, this method will only cast the floating point or complex parameters and buffers to `dtype` (if given). The integral parameters and buffers will be moved `device`, if that is given, but with dtypes unchanged. When `non_blocking` is set, it tries to convert/move asynchronously with respect to the host if possible, e.g., moving CPU Tensors with pinned memory to CUDA devices.

See below for examples.

Note: This method modifies the module in-place.

Parameters

- **device** (`torch.device`) – the desired device of the parameters and buffers in this module
- **dtype** (`torch.dtype`) – the desired floating point or complex dtype of the parameters and buffers in this module
- **tensor** (*torch.Tensor*) – Tensor whose dtype and device are the desired dtype and device for all parameters and buffers in this module
- **memory_format** (`torch.memory_format`) – the desired memory format for 4D parameters and buffers in this module (keyword only argument)

Returns self

Return type Module

Examples:

```
>>> # xdoctest: +IGNORE_WANT("non-deterministic")
>>> linear = nn.Linear(2, 2)
>>> linear.weight
Parameter containing:
tensor([[ 0.1913, -0.3420],
        [-0.5113, -0.2325]])
>>> linear.to(torch.double)
Linear(in_features=2, out_features=2, bias=True)
>>> linear.weight
Parameter containing:
tensor([[ 0.1913, -0.3420],
        [-0.5113, -0.2325]], dtype=torch.float64)
>>> # xdoctest: +REQUIRES(env:TORCH_DOCTEST_CUDA1)
>>> gpu1 = torch.device("cuda:1")
>>> linear.to(gpu1, dtype=torch.half, non_blocking=True)
Linear(in_features=2, out_features=2, bias=True)
>>> linear.weight
Parameter containing:
tensor([[ 0.1914, -0.3420],
        [-0.5112, -0.2324]], dtype=torch.float16, device='cuda:1')
>>> cpu = torch.device("cpu")
>>> linear.to(cpu)
```

(continues on next page)

(continued from previous page)

```
Linear(in_features=2, out_features=2, bias=True)
>>> linear.weight
Parameter containing:
tensor([[ 0.1914, -0.3420],
        [-0.5112, -0.2324]], dtype=torch.float16)

>>> linear = nn.Linear(2, 2, bias=None).to(torch.cdouble)
>>> linear.weight
Parameter containing:
tensor([[ 0.3741+0.j,  0.2382+0.j],
        [ 0.5593+0.j, -0.4443+0.j]], dtype=torch.complex128)
>>> linear(torch.ones(3, 2, dtype=torch.cdouble))
tensor([[0.6122+0.j, 0.1150+0.j],
        [0.6122+0.j, 0.1150+0.j],
        [0.6122+0.j, 0.1150+0.j]], dtype=torch.complex128)
```

to_empty(*, *device: Union[str, torch.device]*) → torch.nn.modules.module.T
Moves the parameters and buffers to the specified device without copying storage.

Parameters **device** (`torch.device`) – The desired device of the parameters and buffers in this module.

Returns self

Return type Module

train(*mode: bool = True*) → torch.nn.modules.module.T
Sets the module in training mode.

This has any effect only on certain modules. See documentations of particular modules for details of their behaviors in training/evaluation mode, if they are affected, e.g. `Dropout`, `BatchNorm`, etc.

Parameters **mode** (*bool*) – whether to set training mode (`True`) or evaluation mode (`False`). Default: `True`.

Returns self

Return type Module

training: bool

type(*dst_type: Union[torch.dtype, str]*) → torch.nn.modules.module.T
Casts all parameters and buffers to `dst_type`.

Note: This method modifies the module in-place.

Parameters **dst_type** (*type or string*) – the desired type

Returns self

Return type Module

xpu(*device: Optional[Union[int, torch.device]] = None*) → torch.nn.modules.module.T
Moves all model parameters and buffers to the XPU.

This also makes associated parameters and buffers different objects. So it should be called before constructing optimizer if the module will live on XPU while being optimized.

Note: This method modifies the module in-place.

Parameters **device** (*int, optional*) – if specified, all parameters will be copied to that device

Returns self

Return type Module

zero_grad(*set_to_none: bool = False*) → None

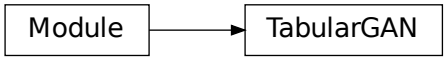
Sets gradients of all model parameters to zero. See similar function under `torch.optim.Optimizer` for more context.

Parameters **set_to_none** (*bool*) – instead of setting to zero, set the grads to None. See `torch.optim.Optimizer.zero_grad()` for details.

synthcity.plugins.core.models.tabular_gan module

class TabularGAN(*X: pandas.core.frame.DataFrame, n_units_latent: int, cond: Optional[Union[pandas.core.frame.DataFrame, pandas.core.series.Series, numpy.ndarray]] = None, generator_n_layers_hidden: int = 2, generator_n_units_hidden: int = 150, generator_nonlin: str = 'leaky_relu', generator_nonlin_out_discrete: str = 'softmax', generator_nonlin_out_continuous: str = 'none', generator_n_iter: int = 1000, generator_batch_norm: bool = False, generator_dropout: float = 0.01, generator_lr: float = 0.001, generator_weight_decay: float = 0.001, generator_opt_betas: tuple = (0.9, 0.999), generator_residual: bool = True, generator_extra_penalties: list = [], discriminator_n_layers_hidden: int = 3, discriminator_n_units_hidden: int = 300, discriminator_nonlin: str = 'leaky_relu', discriminator_n_iter: int = 1, discriminator_batch_norm: bool = False, discriminator_dropout: float = 0.1, discriminator_lr: float = 0.001, discriminator_weight_decay: float = 0.001, discriminator_opt_betas: tuple = (0.9, 0.999), batch_size: int = 64, random_state: int = 0, clipping_value: int = 0, lambda_gradient_penalty: float = 10, lambda_identifiability_penalty: float = 0.1, encoder_max_clusters: int = 20, encoder: Any = None, encoder_whitelist: list = [], dataloader_sampler: Optional[synthcity.utils.samplers.BaseSampler] = None, device: Any = device(type='cpu'), patience: int = 10, patience_metric: Optional[synthcity.metrics.weighted_metrics.WeightedMetrics] = None, n_iter_print: int = 50, n_iter_min: int = 100, adjust_inference_sampling: bool = False, dp_enabled: bool = False, dp_epsilon: float = 3, dp_delta: Optional[float] = None, dp_max_grad_norm: float = 2, dp_secure_mode: bool = False*)

Bases: `torch.nn.modules.module.Module`



GAN for tabular data.

This class combines GAN and tabular encoder to form a generative model for tabular data.

Parameters

- **X** – pd.DataFrame Reference dataset, used for training the tabular encoder
- **n_units_latent** – int Number of latent units
- **cond** – Optional Optional conditional
- **generator_n_layers_hidden** – int Number of hidden layers in the generator
- **generator_n_units_hidden** – int Number of hidden units in each layer of the Generator
- **generator_nonlin** – string, default 'elu' Nonlinearity to use in the generator. Can be 'elu', 'relu', 'selu' or 'leaky_relu'.
- **generator_n_iter** – int Maximum number of iterations in the Generator.
- **generator_batch_norm** – bool Enable/disable batch norm for the generator
- **generator_dropout** – float Dropout value. If 0, the dropout is not used.
- **generator_residual** – bool Use residuals for the generator
- **generator_nonlin_out** – Optional[List[Tuple[str, int]]] List of activations. Useful with the TabularEncoder
- **generator_lr** – float = 2e-4 Generator learning rate, used by the Adam optimizer
- **generator_weight_decay** – float = 1e-3 Generator weight decay, used by the Adam optimizer
- **generator_opt_betas** – tuple = (0.9, 0.999) Generator initial decay rates, used by the Adam Optimizer
- **generator_extra_penalties** – list Additional penalties for the generator. Values: "identifiability_penalty"
- **generator_extra_penalty_cbks** – List[Callable] Additional loss callabacks for the generator. Used by the TabularGAN for the conditional loss
- **discriminator_n_layers_hidden** – int Number of hidden layers in the discriminator
- **discriminator_n_units_hidden** – int Number of hidden units in each layer of the discriminator
- **discriminator_nonlin** – string, default 'relu' Nonlinearity to use in the discriminator. Can be 'elu', 'relu', 'selu' or 'leaky_relu'.
- **discriminator_n_iter** – int Maximum number of iterations in the discriminator.
- **discriminator_batch_norm** – bool Enable/disable batch norm for the discriminator
- **discriminator_dropout** – float Dropout value for the discriminator. If 0, the dropout is not used.
- **discriminator_lr** – float Discriminator learning rate, used by the Adam optimizer
- **discriminator_weight_decay** – float Discriminator weight decay, used by the Adam optimizer
- **discriminator_opt_betas** – tuple Initial weight decays for the Adam optimizer
- **batch_size** – int Batch size
- **n_iter_print** – int Number of iterations after which to print updates and check the validation loss.

- **random_state** – int random_state used
- **n_iter_min** – int Minimum number of iterations to go through before starting early stopping
- **clipping_value** – int, default 0 Gradients clipping value. Zero disables the feature
- **lambda_gradient_penalty** – float = 10 Weight for the gradient penalty
- **lambda_identifiability_penalty** – float = 0.1 Weight for the identifiability penalty, if enabled
- **dataloader_sampler** – Optional[sampler.Sampler] Optional sampler for the dataloader, useful for conditional sampling
- **device** – Any = DEVICE CUDA/CPU
- **adjust_inference_sampling** – bool Adjust the marginal probabilities in the synthetic data to closer match the training set. Active only with the ConditionalSampler
- **settings** (*# privacy*) –
- **dp_enabled** – bool Train the discriminator with Differential Privacy guarantees
- **dp_delta** – Optional[float] Optional DP delta: the probability of information accidentally being leaked. Usually 1 / len(dataset)
- **dp_epsilon** – float = 3 DP epsilon: privacy budget, which is a measure of the amount of privacy that is preserved by a given algorithm. Epsilon is a number that represents the maximum amount of information that an adversary can learn about an individual from the output of a differentially private algorithm. The smaller the value of epsilon, the more private the algorithm is. For example, an algorithm with an epsilon of 0.1 preserves more privacy than an algorithm with an epsilon of 1.0.
- **dp_max_grad_norm** – float max grad norm used for gradient clipping
- **dp_secure_mode** – bool = False, if True uses noise generation approach robust to floating point arithmetic attacks.
- **encoder_max_clusters** – int The max number of clusters to create for continuous columns when encoding
- **encoder** – Pre-trained tabular encoder. If None, a new encoder is trained.
- **encoder_whitelist** – Ignore columns from encoding

T_destination
alias of TypeVar('T_destination', bound=Dict[str, Any])

add_module(*name: str*, *module: Optional[torch.nn.modules.module.Module]*) → None
Adds a child module to the current module.

The module can be accessed as an attribute using the given name.

Parameters

- **name** (*str*) – name of the child module. The child module can be accessed from this module using the given name
- **module** (*Module*) – child module to be added to the module.

adjust_inference_sampling(*enabled: bool*) → None

apply(*fn: Callable[[torch.nn.modules.module.Module], None]*) → torch.nn.modules.module.T
Applies `fn` recursively to every submodule (as returned by `.children()`) as well as self. Typical use includes initializing the parameters of a model (see also nn-init-doc).

Parameters **fn** (Module -> None) – function to be applied to each submodule

Returns self

Return type Module

Example:

```
>>> @torch.no_grad()
>>> def init_weights(m):
>>>     print(m)
>>>     if type(m) == nn.Linear:
>>>         m.weight.fill_(1.0)
>>>         print(m.weight)
>>> net = nn.Sequential(nn.Linear(2, 2), nn.Linear(2, 2))
>>> net.apply(init_weights)
Linear(in_features=2, out_features=2, bias=True)
Parameter containing:
tensor([[1., 1.],
        [1., 1.]], requires_grad=True)
Linear(in_features=2, out_features=2, bias=True)
Parameter containing:
tensor([[1., 1.],
        [1., 1.]], requires_grad=True)
Sequential(
  (0): Linear(in_features=2, out_features=2, bias=True)
  (1): Linear(in_features=2, out_features=2, bias=True)
)
```

bfloat16() → torch.nn.modules.module.T

Casts all floating point parameters and buffers to `bfloat16` datatype.

Note: This method modifies the module in-place.

Returns self

Return type Module

buffers(*recurse: bool = True*) → Iterator[torch.Tensor]

Returns an iterator over module buffers.

Parameters **recurse** (*bool*) – if True, then yields buffers of this module and all submodules. Otherwise, yields only buffers that are direct members of this module.

Yields *torch.Tensor* – module buffer

Example:

```
>>> # xdoctest: +SKIP("undefined vars")
>>> for buf in model.buffers():
>>>     print(type(buf), buf.size())
<class 'torch.Tensor'> (20L,)
<class 'torch.Tensor'> (20L, 1L, 5L, 5L)
```

children() → Iterator[torch.nn.modules.module.Module]

Returns an iterator over immediate children modules.

Yields *Module* – a child module

cpu() → torch.nn.modules.module.T
Moves all model parameters and buffers to the CPU.

Note: This method modifies the module in-place.

Returns self

Return type Module

cuda(*device: Optional[Union[int, torch.device]] = None*) → torch.nn.modules.module.T
Moves all model parameters and buffers to the GPU.

This also makes associated parameters and buffers different objects. So it should be called before constructing optimizer if the module will live on GPU while being optimized.

Note: This method modifies the module in-place.

Parameters **device** (*int, optional*) – if specified, all parameters will be copied to that device

Returns self

Return type Module

decode(*X: pandas.core.frame.DataFrame*) → pandas.core.frame.DataFrame

double() → torch.nn.modules.module.T
Casts all floating point parameters and buffers to `double` datatype.

Note: This method modifies the module in-place.

Returns self

Return type Module

dump_patches: bool = False

encode(*X: pandas.core.frame.DataFrame*) → pandas.core.frame.DataFrame

eval() → torch.nn.modules.module.T
Sets the module in evaluation mode.

This has any effect only on certain modules. See documentations of particular modules for details of their behaviors in training/evaluation mode, if they are affected, e.g. `Dropout`, `BatchNorm`, etc.

This is equivalent with `self.train(False)`.

See locally-disable-grad-doc for a comparison between *.eval()* and several similar mechanisms that may be confused with it.

Returns self

Return type Module

extra_repr() → str
Set the extra representation of the module

To print customized extra information, you should re-implement this method in your own modules. Both single-line and multi-line strings are acceptable.

fit(*X: pandas.core.frame.DataFrame*, *cond: Optional[Union[pandas.core.frame.DataFrame, pandas.core.series.Series, numpy.ndarray]] = None*, *fake_labels_generator: Optional[Callable] = None*, *true_labels_generator: Optional[Callable] = None*, *encoded: bool = False*) → Any

float() → torch.nn.modules.module.T
Casts all floating point parameters and buffers to `float` datatype.

Note: This method modifies the module in-place.

Returns self

Return type Module

forward(*count: int*, *cond: Optional[Union[pandas.core.frame.DataFrame, numpy.ndarray]] = None*) → torch.Tensor
Defines the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

---

generate(*count: int*, *cond: Optional[Union[pandas.core.frame.DataFrame, pandas.core.series.Series, numpy.ndarray]] = None*) → pandas.core.frame.DataFrame

get_buffer(*target: str*) → torch.Tensor
Returns the buffer given by `target` if it exists, otherwise throws an error.

See the docstring for `get_submodule` for a more detailed explanation of this method's functionality as well as how to correctly specify `target`.

Parameters **target** – The fully-qualified string name of the buffer to look for. (See `get_submodule` for how to specify a fully-qualified string.)

Returns The buffer referenced by `target`

Return type torch.Tensor

Raises **AttributeError** – If the target string references an invalid path or resolves to something that is not a buffer

get_encoder() → *synthcity.plugins.core.models.tabular_encoder.TabularEncoder*

get_extra_state() → Any
Returns any extra state to include in the module's state_dict. Implement this and a corresponding *set_extra_state()* for your module if you need to store extra state. This function is called when building the module's *state_dict()*.

Note that extra state should be pickleable to ensure working serialization of the state_dict. We only provide provide backwards compatibility guarantees for serializing Tensors; other objects may break backwards compatibility if their serialized pickled form changes.

Returns Any extra state to store in the module's state_dict

Return type object

`get_parameter`(*target: str*) → torch.nn.parameter.Parameter
Returns the parameter given by `target` if it exists, otherwise throws an error.

See the docstring for `get_submodule` for a more detailed explanation of this method's functionality as well as how to correctly specify `target`.

Parameters `target` – The fully-qualified string name of the Parameter to look for. (See `get_submodule` for how to specify a fully-qualified string.)

Returns The Parameter referenced by `target`

Return type torch.nn.Parameter

Raises `AttributeError` – If the target string references an invalid path or resolves to something that is not an `nn.Parameter`

`get_submodule`(*target: str*) → torch.nn.modules.module.Module
Returns the submodule given by `target` if it exists, otherwise throws an error.

For example, let's say you have an `nn.Module` `A` that looks like this:

```
A(
    (net_b): Module(
        (net_c): Module(
            (conv): Conv2d(16, 33, kernel_size=(3, 3), stride=(2, 2))
        )
        (linear): Linear(in_features=100, out_features=200, bias=True)
    )
)
```

(The diagram shows an `nn.Module` `A`. `A` has a nested submodule `net_b`, which itself has two submodules `net_c` and `linear`. `net_c` then has a submodule `conv`.)

To check whether or not we have the `linear` submodule, we would call `get_submodule("net_b.linear")`. To check whether we have the `conv` submodule, we would call `get_submodule("net_b.net_c.conv")`.

The runtime of `get_submodule` is bounded by the degree of module nesting in `target`. A query against `named_modules` achieves the same result, but it is O(N) in the number of transitive modules. So, for a simple check to see if some submodule exists, `get_submodule` should always be used.

Parameters `target` – The fully-qualified string name of the submodule to look for. (See above example for how to specify a fully-qualified string.)

Returns The submodule referenced by `target`

Return type torch.nn.Module

Raises `AttributeError` – If the target string references an invalid path or resolves to something that is not an `nn.Module`

`half`() → torch.nn.modules.module.T
Casts all floating point parameters and buffers to `half` datatype.

Note: This method modifies the module in-place.

Returns self

Return type Module

ipu(*device: Optional[Union[int, torch.device]] = None*) → torch.nn.modules.module.T
Moves all model parameters and buffers to the IPU.

This also makes associated parameters and buffers different objects. So it should be called before constructing optimizer if the module will live on IPU while being optimized.

Note: This method modifies the module in-place.

Parameters **device** (*int, optional*) – if specified, all parameters will be copied to that device

Returns self

Return type Module

load_state_dict(*state_dict: Mapping[str, Any]*, *strict: bool = True*)
Copies parameters and buffers from *state_dict* into this module and its descendants. If `strict` is `True`, then the keys of *state_dict* must exactly match the keys returned by this module's `state_dict()` function.

Parameters

- **state_dict** (*dict*) – a dict containing parameters and persistent buffers.
- **strict** (*bool, optional*) – whether to strictly enforce that the keys in *state_dict* match the keys returned by this module's `state_dict()` function. Default: `True`

Returns

- **missing_keys** is a list of str containing the missing keys
- **unexpected_keys** is a list of str containing the unexpected keys

Return type `NamedTuple` with `missing_keys` and `unexpected_keys` fields

Note: If a parameter or buffer is registered as `None` and its corresponding key exists in *state_dict*, *load_state_dict()* will raise a `RuntimeError`.

modules() → Iterator[torch.nn.modules.module.Module]
Returns an iterator over all modules in the network.

Yields *Module* – a module in the network

Note: Duplicate modules are returned only once. In the following example, `l` will be returned only once.

Example:

```
>>> l = nn.Linear(2, 2)
>>> net = nn.Sequential(l, l)
>>> for idx, m in enumerate(net.modules()):
...     print(idx, '->', m)

0 -> Sequential(
```

(continues on next page)

(continued from previous page)

```
  (0): Linear(in_features=2, out_features=2, bias=True)
  (1): Linear(in_features=2, out_features=2, bias=True)
)
1 -> Linear(in_features=2, out_features=2, bias=True)
```

named_buffers(*prefix: str = ''*, *recurse: bool = True*) → Iterator[Tuple[str, torch.Tensor]]

Returns an iterator over module buffers, yielding both the name of the buffer as well as the buffer itself.

Parameters

- **prefix** (*str*) – prefix to prepend to all buffer names.
- **recurse** (*bool*) – if True, then yields buffers of this module and all submodules. Otherwise, yields only buffers that are direct members of this module.

Yields *(str, torch.Tensor)* – Tuple containing the name and buffer

Example:

```
>>> # xdoctest: +SKIP("undefined vars")
>>> for name, buf in self.named_buffers():
>>>    if name in ['running_var']:
>>>        print(buf.size())
```

named_children() → Iterator[Tuple[str, torch.nn.modules.module.Module]]

Returns an iterator over immediate children modules, yielding both the name of the module as well as the module itself.

Yields *(str, Module)* – Tuple containing a name and child module

Example:

```
>>> # xdoctest: +SKIP("undefined vars")
>>> for name, module in model.named_children():
>>>     if name in ['conv4', 'conv5']:
>>>         print(module)
```

named_modules(*memo: Optional[Set[torch.nn.modules.module.Module]] = None*, *prefix: str = ''*, *remove_duplicate: bool = True*)

Returns an iterator over all modules in the network, yielding both the name of the module as well as the module itself.

Parameters

- **memo** – a memo to store the set of modules already added to the result
- **prefix** – a prefix that will be added to the name of the module
- **remove_duplicate** – whether to remove the duplicated module instances in the result or not

Yields *(str, Module)* – Tuple of name and module

Note: Duplicate modules are returned only once. In the following example, `l` will be returned only once.

Example:

```
>>> l = nn.Linear(2, 2)
>>> net = nn.Sequential(l, l)
>>> for idx, m in enumerate(net.named_modules()):
...     print(idx, '->', m)

0 -> ('', Sequential(
  (0): Linear(in_features=2, out_features=2, bias=True)
  (1): Linear(in_features=2, out_features=2, bias=True)
))
1 -> ('0', Linear(in_features=2, out_features=2, bias=True))
```

named_parameters(*prefix: str = ''*, *recurse: bool = True*) → Iterator[Tuple[str, torch.nn.parameter.Parameter]]

Returns an iterator over module parameters, yielding both the name of the parameter as well as the parameter itself.

Parameters

- **prefix** (*str*) – prefix to prepend to all parameter names.
- **recurse** (*bool*) – if True, then yields parameters of this module and all submodules. Otherwise, yields only parameters that are direct members of this module.

Yields *(str, Parameter)* – Tuple containing the name and parameter

Example:

```
>>> # xdoctest: +SKIP("undefined vars")
>>> for name, param in self.named_parameters():
>>>    if name in ['bias']:
>>>        print(param.size())
```

parameters(*recurse: bool = True*) → Iterator[torch.nn.parameter.Parameter]

Returns an iterator over module parameters.

This is typically passed to an optimizer.

Parameters **recurse** (*bool*) – if True, then yields parameters of this module and all submodules. Otherwise, yields only parameters that are direct members of this module.

Yields *Parameter* – module parameter

Example:

```
>>> # xdoctest: +SKIP("undefined vars")
>>> for param in model.parameters():
>>>     print(type(param), param.size())
<class 'torch.Tensor'> (20L,)
<class 'torch.Tensor'> (20L, 1L, 5L, 5L)
```

register_backward_hook(*hook: Callable[[torch.nn.modules.module.Module, Union[Tuple[torch.Tensor, ...], torch.Tensor], Union[Tuple[torch.Tensor, ...], torch.Tensor]], Union[None, torch.Tensor]]*) → torch.utils.hooks.RemovableHandle

Registers a backward hook on the module.

This function is deprecated in favor of `register_full_backward_hook()` and the behavior of this function will change in future versions.

Returns a handle that can be used to remove the added hook by calling `handle.remove()`

Return type torch.utils.hooks.RemovableHandle

register_buffer(*name: str*, *tensor: Optional[torch.Tensor]*, *persistent: bool = True*) → None
Adds a buffer to the module.

This is typically used to register a buffer that should not to be considered a model parameter. For example, BatchNorm's `running_mean` is not a parameter, but is part of the module's state. Buffers, by default, are persistent and will be saved alongside parameters. This behavior can be changed by setting `persistent` to `False`. The only difference between a persistent buffer and a non-persistent buffer is that the latter will not be a part of this module's *state_dict*.

Buffers can be accessed as attributes using given names.

Parameters

- **name** (*str*) – name of the buffer. The buffer can be accessed from this module using the given name
- **tensor** (*Tensor or None*) – buffer to be registered. If `None`, then operations that run on buffers, such as *cuda*, are ignored. If `None`, the buffer is **not** included in the module's *state_dict*.
- **persistent** (*bool*) – whether the buffer is part of this module's *state_dict*.

Example:

```
>>> # xdoctest: +SKIP("undefined vars")
>>> self.register_buffer('running_mean', torch.zeros(num_features))
```

register_forward_hook(*hook: Callable[[...], None]*) → torch.utils.hooks.RemovableHandle
Registers a forward hook on the module.

The hook will be called every time after *forward()* has computed an output. It should have the following signature:

```
hook(module, input, output) -> None or modified output
```

The input contains only the positional arguments given to the module. Keyword arguments won't be passed to the hooks and only to the `forward`. The hook can modify the output. It can modify the input inplace but it will not have effect on forward since this is called after *forward()* is called.

Returns a handle that can be used to remove the added hook by calling `handle.remove()`

Return type torch.utils.hooks.RemovableHandle

register_forward_pre_hook(*hook: Callable[[...], None]*) → torch.utils.hooks.RemovableHandle
Registers a forward pre-hook on the module.

The hook will be called every time before *forward()* is invoked. It should have the following signature:

```
hook(module, input) -> None or modified input
```

The input contains only the positional arguments given to the module. Keyword arguments won't be passed to the hooks and only to the `forward`. The hook can modify the input. User can either return a tuple or a single modified value in the hook. We will wrap the value into a tuple if a single value is returned(unless that value is already a tuple).

Returns a handle that can be used to remove the added hook by calling `handle.remove()`

Return type torch.utils.hooks.RemovableHandle

register_full_backward_hook(*hook: Callable[[torch.nn.modules.module.Module, Union[Tuple[torch.Tensor, ...], torch.Tensor], Union[Tuple[torch.Tensor, ...], torch.Tensor]], Union[None, torch.Tensor]]*) → torch.utils.hooks.RemovableHandle

Registers a backward hook on the module.

The hook will be called every time the gradients with respect to a module are computed, i.e. the hook will execute if and only if the gradients with respect to module outputs are computed. The hook should have the following signature:

```
hook(module, grad_input, grad_output) -> tuple(Tensor) or None
```

The `grad_input` and `grad_output` are tuples that contain the gradients with respect to the inputs and outputs respectively. The hook should not modify its arguments, but it can optionally return a new gradient with respect to the input that will be used in place of `grad_input` in subsequent computations. `grad_input` will only correspond to the inputs given as positional arguments and all kwarg arguments are ignored. Entries in `grad_input` and `grad_output` will be `None` for all non-Tensor arguments.

For technical reasons, when this hook is applied to a Module, its forward function will receive a view of each Tensor passed to the Module. Similarly the caller will receive a view of each Tensor returned by the Module's forward function.

Warning: Modifying inputs or outputs inplace is not allowed when using backward hooks and will raise an error.

Returns a handle that can be used to remove the added hook by calling `handle.remove()`

Return type `torch.utils.hooks.RemovableHandle`

register_load_state_dict_post_hook(*hook*)

Registers a post hook to be run after module's `load_state_dict` is called.

It should have the following signature:: hook(module, incompatible_keys) -> None

The `module` argument is the current module that this hook is registered on, and the `incompatible_keys` argument is a `NamedTuple` consisting of attributes `missing_keys` and `unexpected_keys`. `missing_keys` is a `list` of `str` containing the missing keys and `unexpected_keys` is a `list` of `str` containing the unexpected keys.

The given incompatible_keys can be modified inplace if needed.

Note that the checks performed when calling *load_state_dict()* with `strict=True` are affected by modifications the hook makes to `missing_keys` or `unexpected_keys`, as expected. Additions to either set of keys will result in an error being thrown when `strict=True`, and clearning out both missing and unexpected keys will avoid an error.

Returns a handle that can be used to remove the added hook by calling `handle.remove()`

Return type `torch.utils.hooks.RemovableHandle`

register_module(*name: str*, *module: Optional[torch.nn.modules.module.Module]*) → None

Alias for *add_module()*.

register_parameter(*name: str*, *param: Optional[torch.nn.parameter.Parameter]*) → None

Adds a parameter to the module.

The parameter can be accessed as an attribute using given name.

Parameters

- **name** (*str*) – name of the parameter. The parameter can be accessed from this module using the given name
- **param** (*Parameter or None*) – parameter to be added to the module. If `None`, then operations that run on parameters, such as *cuda*, are ignored. If `None`, the parameter is **not** included in the module's *state_dict*.

requires_grad_(*requires_grad: bool = True*) → torch.nn.modules.module.T
Change if autograd should record operations on parameters in this module.

This method sets the parameters' `requires_grad` attributes in-place.

This method is helpful for freezing part of the module for finetuning or training parts of a model individually (e.g., GAN training).

See locally-disable-grad-doc for a comparison between *.requires_grad_()* and several similar mechanisms that may be confused with it.

Parameters **requires_grad** (*bool*) – whether autograd should record operations on parameters in this module. Default: `True`.

Returns self

Return type Module

set_extra_state(*state: Any*)
This function is called from *load_state_dict()* to handle any extra state found within the *state_dict*. Implement this function and a corresponding *get_extra_state()* for your module if you need to store extra state within its *state_dict*.

Parameters **state** (*dict*) – Extra state from the *state_dict*

share_memory() → torch.nn.modules.module.T
See `torch.Tensor.share_memory_()`

state_dict(**args*, *destination=None*, *prefix=''*, *keep_vars=False*)
Returns a dictionary containing references to the whole state of the module.

Both parameters and persistent buffers (e.g. running averages) are included. Keys are corresponding parameter and buffer names. Parameters and buffers set to `None` are not included.

Note: The returned object is a shallow copy. It contains references to the module's parameters and buffers.

Warning: Currently `state_dict()` also accepts positional arguments for `destination`, `prefix` and `keep_vars` in order. However, this is being deprecated and keyword arguments will be enforced in future releases.

Warning: Please avoid the use of argument `destination` as it is not designed for end-users.

Parameters

- **destination** (*dict, optional*) – If provided, the state of module will be updated into the dict and the same object is returned. Otherwise, an `OrderedDict` will be created and returned. Default: `None`.
- **prefix** (*str, optional*) – a prefix added to parameter and buffer names to compose the keys in state_dict. Default: `''`.

- **keep_vars** (*bool, optional*) – by default the `Tensor` s returned in the state dict are detached from autograd. If it's set to `True`, detaching will not be performed. Default: `False`.

Returns a dictionary containing a whole state of the module

Return type dict

Example:

```
>>> # xdoctest: +SKIP("undefined vars")
>>> module.state_dict().keys()
['bias', 'weight']
```

to(**args*, ***kwargs*)

Moves and/or casts the parameters and buffers.

This can be called as

to(*device=None*, *dtype=None*, *non_blocking=False*)

to(*dtype*, *non_blocking=False*)

to(*tensor*, *non_blocking=False*)

to(*memory_format=torch.channels_last*)

Its signature is similar to `torch.Tensor.to()`, but only accepts floating point or complex `dtypes`. In addition, this method will only cast the floating point or complex parameters and buffers to `dtype` (if given). The integral parameters and buffers will be moved `device`, if that is given, but with dtypes unchanged. When `non_blocking` is set, it tries to convert/move asynchronously with respect to the host if possible, e.g., moving CPU Tensors with pinned memory to CUDA devices.

See below for examples.

Note: This method modifies the module in-place.

Parameters

- **device** (`torch.device`) – the desired device of the parameters and buffers in this module
- **dtype** (`torch.dtype`) – the desired floating point or complex dtype of the parameters and buffers in this module
- **tensor** (*torch.Tensor*) – Tensor whose dtype and device are the desired dtype and device for all parameters and buffers in this module
- **memory_format** (`torch.memory_format`) – the desired memory format for 4D parameters and buffers in this module (keyword only argument)

Returns self

Return type Module

Examples:

```
>>> # xdoctest: +IGNORE_WANT("non-deterministic")
>>> linear = nn.Linear(2, 2)
>>> linear.weight
Parameter containing:
tensor([[ 0.1913, -0.3420],
        [-0.5113, -0.2325]])
>>> linear.to(torch.double)
Linear(in_features=2, out_features=2, bias=True)
>>> linear.weight
Parameter containing:
tensor([[ 0.1913, -0.3420],
        [-0.5113, -0.2325]], dtype=torch.float64)
>>> # xdoctest: +REQUIRES(env:TORCH_DOCTEST_CUDA1)
>>> gpu1 = torch.device("cuda:1")
>>> linear.to(gpu1, dtype=torch.half, non_blocking=True)
Linear(in_features=2, out_features=2, bias=True)
>>> linear.weight
Parameter containing:
tensor([[ 0.1914, -0.3420],
        [-0.5112, -0.2324]], dtype=torch.float16, device='cuda:1')
>>> cpu = torch.device("cpu")
>>> linear.to(cpu)
Linear(in_features=2, out_features=2, bias=True)
>>> linear.weight
Parameter containing:
tensor([[ 0.1914, -0.3420],
        [-0.5112, -0.2324]], dtype=torch.float16)

>>> linear = nn.Linear(2, 2, bias=None).to(torch.cdouble)
>>> linear.weight
Parameter containing:
tensor([[ 0.3741+0.j,  0.2382+0.j],
        [ 0.5593+0.j, -0.4443+0.j]], dtype=torch.complex128)
>>> linear(torch.ones(3, 2, dtype=torch.cdouble))
tensor([[0.6122+0.j, 0.1150+0.j],
        [0.6122+0.j, 0.1150+0.j],
        [0.6122+0.j, 0.1150+0.j]], dtype=torch.complex128)
```

to_empty(*, *device: Union[str, torch.device]*) → torch.nn.modules.module.T
Moves the parameters and buffers to the specified device without copying storage.

Parameters **device** (torch.device) – The desired device of the parameters and buffers in this module.

Returns self

Return type Module

train(*mode: bool = True*) → torch.nn.modules.module.T
Sets the module in training mode.

This has any effect only on certain modules. See documentations of particular modules for details of their behaviors in training/evaluation mode, if they are affected, e.g. Dropout, BatchNorm, etc.

Parameters **mode** (*bool*) – whether to set training mode (True) or evaluation mode (False). Default: True.

Returns self

Return type Module

`training: bool`

`type`(*dst_type: Union[torch.dtype, str]*) → torch.nn.modules.module.T
Casts all parameters and buffers to `dst_type`.

Note: This method modifies the module in-place.

Parameters `dst_type` (*type or string*) – the desired type

Returns self

Return type Module

`xpu`(*device: Optional[Union[int, torch.device]] = None*) → torch.nn.modules.module.T
Moves all model parameters and buffers to the XPU.

This also makes associated parameters and buffers different objects. So it should be called before constructing optimizer if the module will live on XPU while being optimized.

Note: This method modifies the module in-place.

Parameters `device` (*int, optional*) – if specified, all parameters will be copied to that device

Returns self

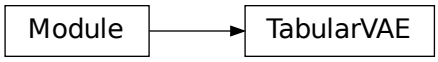
Return type Module

`zero_grad`(*set_to_none: bool = False*) → None
Sets gradients of all model parameters to zero. See similar function under `torch.optim.Optimizer` for more context.

Parameters `set_to_none` (*bool*) – instead of setting to zero, set the grads to None. See `torch.optim.Optimizer.zero_grad()` for details.

synthcity.plugins.core.models.tabular_vae module

`class TabularVAE`(*X: pandas.core.frame.DataFrame*, *n_units_embedding: int*, *cond: Optional[Union[pandas.core.frame.DataFrame, pandas.core.series.Series, numpy.ndarray]] = None*, *lr: float = 0.0002*, *n_iter: int = 500*, *weight_decay: float = 0.001*, *batch_size: int = 64*, *random_state: int = 0*, *loss_strategy: str = 'standard'*, *encoder_max_clusters: int = 20*, *decoder_n_layers_hidden: int = 2*, *decoder_n_units_hidden: int = 250*, *decoder_nonlin: str = 'leaky_relu'*, *decoder_nonlin_out_discrete: str = 'softmax'*, *decoder_nonlin_out_continuous: str = 'tanh'*, *decoder_batch_norm: bool = False*, *decoder_dropout: float = 0*, *decoder_residual: bool = True*, *encoder_n_layers_hidden: int = 3*, *encoder_n_units_hidden: int = 300*, *encoder_nonlin: str = 'leaky_relu'*, *encoder_batch_norm: bool = False*, *encoder_dropout: float = 0.1*, *encoder_whitelist: list = []*, *device: Any = device(type='cpu')*, *robust_divergence_beta: int = 2*, *loss_factor: int = 1*, *dataloader_sampler: Optional[synthcity.utils.samplers.BaseSampler] = None*, *clipping_value: int = 1*, *n_iter_min: int = 100*, *n_iter_print: int = 10*, *patience: int = 20*)
Bases: `torch.nn.modules.module.Module`



VAE for tabular data.

This class combines VAE and tabular encoder to form a generative model for tabular data.

Parameters

- **X** – pd.DataFrame Reference dataset, used for training the tabular encoder
- **cond** – Optional Optional conditional
- **decoder_n_layers_hidden** – int Number of hidden layers in the decoder
- **decoder_n_units_hidden** – int Number of hidden units in each layer of the decoder
- **decoder_nonlin** – string, default 'elu' Nonlinearity to use in the decoder. Can be 'elu', 'relu', 'selu' or 'leaky_relu'.
- **decoder_n_iter** – int Maximum number of iterations in the decoder.
- **decoder_batch_norm** – bool Enable/disable batch norm for the decoder
- **decoder_dropout** – float Dropout value. If 0, the dropout is not used.
- **decoder_residual** – bool Use residuals for the decoder
- **encoder_n_layers_hidden** – int Number of hidden layers in the encoder
- **encoder_n_units_hidden** – int Number of hidden units in each layer of the encoder
- **encoder_nonlin** – string, default 'relu' Nonlinearity to use in the encoder. Can be 'elu', 'relu', 'selu' or 'leaky_relu'.
- **encoder_n_iter** – int Maximum number of iterations in the encoder.
- **encoder_batch_norm** – bool Enable/disable batch norm for the encoder
- **encoder_dropout** – float Dropout value for the encoder. If 0, the dropout is not used.
- **lr** – float learning rate for optimizer.
- **weight_decay** – float l2 (ridge) penalty for the weights.
- **batch_size** – int Batch size
- **random_state** – int random_state used
- **encoder_max_clusters** – int The max number of clusters to create for continuous columns when encoding
- **stopping** (*# early*) –
- **n_iter_print** – int Number of iterations after which to print updates and check the validation loss.
- **n_iter_min** – int Minimum number of iterations to go through before starting early stopping

- **patience** – int Max number of iterations without any improvement before early stopping is trigged.

T_destination
alias of TypeVar('T_destination', bound=Dict[str, Any])

add_module(*name: str*, *module: Optional[torch.nn.modules.module.Module]*) → None
Adds a child module to the current module.

The module can be accessed as an attribute using the given name.

Parameters

- **name** (*str*) – name of the child module. The child module can be accessed from this module using the given name
- **module** (*Module*) – child module to be added to the module.

apply(*fn: Callable[[torch.nn.modules.module.Module], None]*) → torch.nn.modules.module.T
Applies `fn` recursively to every submodule (as returned by `.children()`) as well as self. Typical use includes initializing the parameters of a model (see also nn-init-doc).

Parameters **fn** (`Module` -> None) – function to be applied to each submodule

Returns self

Return type Module

Example:

```
>>> @torch.no_grad()
>>> def init_weights(m):
>>>     print(m)
>>>     if type(m) == nn.Linear:
>>>         m.weight.fill_(1.0)
>>>         print(m.weight)
>>> net = nn.Sequential(nn.Linear(2, 2), nn.Linear(2, 2))
>>> net.apply(init_weights)
Linear(in_features=2, out_features=2, bias=True)
Parameter containing:
tensor([[1., 1.],
        [1., 1.]], requires_grad=True)
Linear(in_features=2, out_features=2, bias=True)
Parameter containing:
tensor([[1., 1.],
        [1., 1.]], requires_grad=True)
Sequential(
  (0): Linear(in_features=2, out_features=2, bias=True)
  (1): Linear(in_features=2, out_features=2, bias=True)
)
```

bfloat16() → torch.nn.modules.module.T
Casts all floating point parameters and buffers to `bfloat16` datatype.

Note: This method modifies the module in-place.

Returns self

Return type Module

buffers(*recurse: bool = True*) → Iterator[torch.Tensor]
Returns an iterator over module buffers.

Parameters **recurse** (*bool*) – if True, then yields buffers of this module and all submodules. Otherwise, yields only buffers that are direct members of this module.

Yields *torch.Tensor* – module buffer

Example:

```
>>> # xdoctest: +SKIP("undefined vars")
>>> for buf in model.buffers():
>>>     print(type(buf), buf.size())
<class 'torch.Tensor'> (20L,)
<class 'torch.Tensor'> (20L, 1L, 5L, 5L)
```

children() → Iterator[torch.nn.modules.module.Module]
Returns an iterator over immediate children modules.

Yields *Module* – a child module

cpu() → torch.nn.modules.module.T
Moves all model parameters and buffers to the CPU.

Note: This method modifies the module in-place.

Returns self

Return type Module

cuda(*device: Optional[Union[int, torch.device]] = None*) → torch.nn.modules.module.T
Moves all model parameters and buffers to the GPU.

This also makes associated parameters and buffers different objects. So it should be called before constructing optimizer if the module will live on GPU while being optimized.

Note: This method modifies the module in-place.

Parameters **device** (*int, optional*) – if specified, all parameters will be copied to that device

Returns self

Return type Module

decode(*X: pandas.core.frame.DataFrame*) → pandas.core.frame.DataFrame

double() → torch.nn.modules.module.T
Casts all floating point parameters and buffers to `double` datatype.

Note: This method modifies the module in-place.

Returns self

Return type Module

dump_patches: bool = False

encode(*X: pandas.core.frame.DataFrame*) → pandas.core.frame.DataFrame

eval() → torch.nn.modules.module.T
Sets the module in evaluation mode.

This has any effect only on certain modules. See documentations of particular modules for details of their behaviors in training/evaluation mode, if they are affected, e.g. `Dropout`, `BatchNorm`, etc.

This is equivalent with `self.train(False)`.

See locally-disable-grad-doc for a comparison between *.eval()* and several similar mechanisms that may be confused with it.

Returns self

Return type Module

extra_repr() → str
Set the extra representation of the module

To print customized extra information, you should re-implement this method in your own modules. Both single-line and multi-line strings are acceptable.

fit(*X: pandas.core.frame.DataFrame*, *cond: Optional[Union[pandas.core.frame.DataFrame, pandas.core.series.Series, numpy.ndarray]] = None*, ***kwargs: Any*) → Any

float() → torch.nn.modules.module.T
Casts all floating point parameters and buffers to `float` datatype.

Note: This method modifies the module in-place.

Returns self

Return type Module

forward(*count: int*, *cond: Optional[Union[pandas.core.frame.DataFrame, pandas.core.series.Series, numpy.ndarray]] = None*) → torch.Tensor
Defines the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

---

generate(*count: int*, *cond: Optional[Union[pandas.core.frame.DataFrame, pandas.core.series.Series, numpy.ndarray]] = None*, ***kwargs: Any*) → pandas.core.frame.DataFrame

get_buffer(*target: str*) → torch.Tensor
Returns the buffer given by `target` if it exists, otherwise throws an error.

See the docstring for `get_submodule` for a more detailed explanation of this method's functionality as well as how to correctly specify `target`.

Parameters **target** – The fully-qualified string name of the buffer to look for. (See `get_submodule` for how to specify a fully-qualified string.)

Returns The buffer referenced by `target`

Return type torch.Tensor

Raises **AttributeError** – If the target string references an invalid path or resolves to something that is not a buffer

get_encoder() → *synthcity.plugins.core.models.tabular_encoder.TabularEncoder*

get_extra_state() → Any

Returns any extra state to include in the module's state_dict. Implement this and a corresponding *set_extra_state()* for your module if you need to store extra state. This function is called when building the module's *state_dict()*.

Note that extra state should be pickleable to ensure working serialization of the state_dict. We only provide provide backwards compatibility guarantees for serializing Tensors; other objects may break backwards compatibility if their serialized pickled form changes.

Returns Any extra state to store in the module's state_dict

Return type object

get_parameter(*target: str*) → torch.nn.parameter.Parameter

Returns the parameter given by `target` if it exists, otherwise throws an error.

See the docstring for `get_submodule` for a more detailed explanation of this method's functionality as well as how to correctly specify `target`.

Parameters **target** – The fully-qualified string name of the Parameter to look for. (See `get_submodule` for how to specify a fully-qualified string.)

Returns The Parameter referenced by `target`

Return type torch.nn.Parameter

Raises **AttributeError** – If the target string references an invalid path or resolves to something that is not an `nn.Parameter`

get_submodule(*target: str*) → torch.nn.modules.module.Module

Returns the submodule given by `target` if it exists, otherwise throws an error.

For example, let's say you have an `nn.Module` `A` that looks like this:

```
A(
    (net_b): Module(
        (net_c): Module(
            (conv): Conv2d(16, 33, kernel_size=(3, 3), stride=(2, 2))
        )
        (linear): Linear(in_features=100, out_features=200, bias=True)
    )
)
```

(The diagram shows an `nn.Module` `A`. `A` has a nested submodule `net_b`, which itself has two submodules `net_c` and `linear`. `net_c` then has a submodule `conv`.)

To check whether or not we have the `linear` submodule, we would call `get_submodule("net_b.linear")`. To check whether we have the `conv` submodule, we would call `get_submodule("net_b.net_c.conv")`.

The runtime of `get_submodule` is bounded by the degree of module nesting in `target`. A query against `named_modules` achieves the same result, but it is O(N) in the number of transitive modules. So, for a simple check to see if some submodule exists, `get_submodule` should always be used.

Parameters **target** – The fully-qualified string name of the submodule to look for. (See above example for how to specify a fully-qualified string.)

Returns The submodule referenced by `target`

Return type torch.nn.Module

Raises **AttributeError** – If the target string references an invalid path or resolves to something that is not an `nn.Module`

half() → torch.nn.modules.module.T
Casts all floating point parameters and buffers to `half` datatype.

Note: This method modifies the module in-place.

Returns self

Return type Module

ipu(*device: Optional[Union[int, torch.device]] = None*) → torch.nn.modules.module.T
Moves all model parameters and buffers to the IPU.

This also makes associated parameters and buffers different objects. So it should be called before constructing optimizer if the module will live on IPU while being optimized.

Note: This method modifies the module in-place.

Parameters **device** (*int, optional*) – if specified, all parameters will be copied to that device

Returns self

Return type Module

load_state_dict(*state_dict: Mapping[str, Any]*, *strict: bool = True*)
Copies parameters and buffers from *state_dict* into this module and its descendants. If `strict` is `True`, then the keys of *state_dict* must exactly match the keys returned by this module's `state_dict()` function.

Parameters

- **state_dict** (*dict*) – a dict containing parameters and persistent buffers.
- **strict** (*bool, optional*) – whether to strictly enforce that the keys in *state_dict* match the keys returned by this module's `state_dict()` function. Default: `True`

Returns

- **missing_keys** is a list of str containing the missing keys
- **unexpected_keys** is a list of str containing the unexpected keys

Return type `NamedTuple` with `missing_keys` and `unexpected_keys` fields

Note: If a parameter or buffer is registered as `None` and its corresponding key exists in *state_dict*, *load_state_dict()* will raise a `RuntimeError`.

---

modules() → Iterator[torch.nn.modules.module.Module]
Returns an iterator over all modules in the network.

Yields *Module* – a module in the network

Note: Duplicate modules are returned only once. In the following example, `l` will be returned only once.

Example:

```
>>> l = nn.Linear(2, 2)
>>> net = nn.Sequential(l, l)
>>> for idx, m in enumerate(net.modules()):
...     print(idx, '->', m)

0 -> Sequential(
  (0): Linear(in_features=2, out_features=2, bias=True)
  (1): Linear(in_features=2, out_features=2, bias=True)
)
1 -> Linear(in_features=2, out_features=2, bias=True)
```

named_buffers(*prefix: str = ''*, *recurse: bool = True*) → Iterator[Tuple[str, torch.Tensor]]

Returns an iterator over module buffers, yielding both the name of the buffer as well as the buffer itself.

Parameters

- **prefix** (*str*) – prefix to prepend to all buffer names.
- **recurse** (*bool*) – if True, then yields buffers of this module and all submodules. Otherwise, yields only buffers that are direct members of this module.

Yields *(str, torch.Tensor)* – Tuple containing the name and buffer

Example:

```
>>> # xdoctest: +SKIP("undefined vars")
>>> for name, buf in self.named_buffers():
>>>    if name in ['running_var']:
>>>        print(buf.size())
```

named_children() → Iterator[Tuple[str, torch.nn.modules.module.Module]]

Returns an iterator over immediate children modules, yielding both the name of the module as well as the module itself.

Yields *(str, Module)* – Tuple containing a name and child module

Example:

```
>>> # xdoctest: +SKIP("undefined vars")
>>> for name, module in model.named_children():
>>>     if name in ['conv4', 'conv5']:
>>>         print(module)
```

named_modules(*memo: Optional[Set[torch.nn.modules.module.Module]] = None*, *prefix: str = ''*, *remove_duplicate: bool = True*)

Returns an iterator over all modules in the network, yielding both the name of the module as well as the module itself.

Parameters

- **memo** – a memo to store the set of modules already added to the result
- **prefix** – a prefix that will be added to the name of the module

- **remove_duplicate** – whether to remove the duplicated module instances in the result or not

Yields *(str, Module)* – Tuple of name and module

Note: Duplicate modules are returned only once. In the following example, `l` will be returned only once.

Example:

```
>>> l = nn.Linear(2, 2)
>>> net = nn.Sequential(l, l)
>>> for idx, m in enumerate(net.named_modules()):
...     print(idx, '->', m)

0 -> ('', Sequential(
  (0): Linear(in_features=2, out_features=2, bias=True)
  (1): Linear(in_features=2, out_features=2, bias=True)
))
1 -> ('0', Linear(in_features=2, out_features=2, bias=True))
```

named_parameters(*prefix: str = ''*, *recurse: bool = True*) → Iterator[Tuple[str, torch.nn.parameter.Parameter]]

Returns an iterator over module parameters, yielding both the name of the parameter as well as the parameter itself.

Parameters

- **prefix** (*str*) – prefix to prepend to all parameter names.
- **recurse** (*bool*) – if True, then yields parameters of this module and all submodules. Otherwise, yields only parameters that are direct members of this module.

Yields *(str, Parameter)* – Tuple containing the name and parameter

Example:

```
>>> # xdoctest: +SKIP("undefined vars")
>>> for name, param in self.named_parameters():
>>>    if name in ['bias']:
>>>        print(param.size())
```

parameters(*recurse: bool = True*) → Iterator[torch.nn.parameter.Parameter]

Returns an iterator over module parameters.

This is typically passed to an optimizer.

Parameters **recurse** (*bool*) – if True, then yields parameters of this module and all submodules. Otherwise, yields only parameters that are direct members of this module.

Yields *Parameter* – module parameter

Example:

```
>>> # xdoctest: +SKIP("undefined vars")
>>> for param in model.parameters():
>>>     print(type(param), param.size())
<class 'torch.Tensor'> (20L,)
<class 'torch.Tensor'> (20L, 1L, 5L, 5L)
```

register_backward_hook(*hook: Callable[[torch.nn.modules.module.Module, Union[Tuple[torch.Tensor, ...], torch.Tensor], Union[Tuple[torch.Tensor, ...], torch.Tensor]], Union[None, torch.Tensor]]*) → torch.utils.hooks.RemovableHandle

Registers a backward hook on the module.

This function is deprecated in favor of `register_full_backward_hook()` and the behavior of this function will change in future versions.

Returns a handle that can be used to remove the added hook by calling `handle.remove()`

Return type `torch.utils.hooks.RemovableHandle`

register_buffer(*name: str*, *tensor: Optional[torch.Tensor]*, *persistent: bool = True*) → None

Adds a buffer to the module.

This is typically used to register a buffer that should not to be considered a model parameter. For example, BatchNorm's `running_mean` is not a parameter, but is part of the module's state. Buffers, by default, are persistent and will be saved alongside parameters. This behavior can be changed by setting `persistent` to `False`. The only difference between a persistent buffer and a non-persistent buffer is that the latter will not be a part of this module's *state_dict*.

Buffers can be accessed as attributes using given names.

Parameters

- **name** (*str*) – name of the buffer. The buffer can be accessed from this module using the given name
- **tensor** (*Tensor or None*) – buffer to be registered. If `None`, then operations that run on buffers, such as *cuda*, are ignored. If `None`, the buffer is **not** included in the module's *state_dict*.
- **persistent** (*bool*) – whether the buffer is part of this module's *state_dict*.

Example:

```
>>> # xdoctest: +SKIP("undefined vars")
>>> self.register_buffer('running_mean', torch.zeros(num_features))
```

register_forward_hook(*hook: Callable[[...], None]*) → torch.utils.hooks.RemovableHandle

Registers a forward hook on the module.

The hook will be called every time after *forward()* has computed an output. It should have the following signature:

```
hook(module, input, output) -> None or modified output
```

The input contains only the positional arguments given to the module. Keyword arguments won't be passed to the hooks and only to the `forward`. The hook can modify the output. It can modify the input inplace but it will not have effect on forward since this is called after *forward()* is called.

Returns a handle that can be used to remove the added hook by calling `handle.remove()`

Return type `torch.utils.hooks.RemovableHandle`

register_forward_pre_hook(*hook: Callable[[...], None]*) → torch.utils.hooks.RemovableHandle

Registers a forward pre-hook on the module.

The hook will be called every time before *forward()* is invoked. It should have the following signature:

```
hook(module, input) -> None or modified input
```

The input contains only the positional arguments given to the module. Keyword arguments won't be passed to the hooks and only to the `forward`. The hook can modify the input. User can either return a tuple or a single modified value in the hook. We will wrap the value into a tuple if a single value is returned(unless that value is already a tuple).

Returns a handle that can be used to remove the added hook by calling `handle.remove()`

Return type `torch.utils.hooks.RemovableHandle`

register_full_backward_hook(*hook: Callable[[torch.nn.modules.module.Module, Union[Tuple[torch.Tensor, ...], torch.Tensor], Union[Tuple[torch.Tensor, ...], torch.Tensor]], Union[None, torch.Tensor]]*) → torch.utils.hooks.RemovableHandle

Registers a backward hook on the module.

The hook will be called every time the gradients with respect to a module are computed, i.e. the hook will execute if and only if the gradients with respect to module outputs are computed. The hook should have the following signature:

```
hook(module, grad_input, grad_output) -> tuple(Tensor) or None
```

The `grad_input` and `grad_output` are tuples that contain the gradients with respect to the inputs and outputs respectively. The hook should not modify its arguments, but it can optionally return a new gradient with respect to the input that will be used in place of `grad_input` in subsequent computations. `grad_input` will only correspond to the inputs given as positional arguments and all kwarg arguments are ignored. Entries in `grad_input` and `grad_output` will be `None` for all non-Tensor arguments.

For technical reasons, when this hook is applied to a Module, its forward function will receive a view of each Tensor passed to the Module. Similarly the caller will receive a view of each Tensor returned by the Module's forward function.

Warning: Modifying inputs or outputs inplace is not allowed when using backward hooks and will raise an error.

Returns a handle that can be used to remove the added hook by calling `handle.remove()`

Return type `torch.utils.hooks.RemovableHandle`

register_load_state_dict_post_hook(*hook*)

Registers a post hook to be run after module's `load_state_dict` is called.

It should have the following signature:: hook(module, incompatible_keys) -> None

The `module` argument is the current module that this hook is registered on, and the `incompatible_keys` argument is a `NamedTuple` consisting of attributes `missing_keys` and `unexpected_keys`. `missing_keys` is a `list` of `str` containing the missing keys and `unexpected_keys` is a `list` of `str` containing the unexpected keys.

The given incompatible_keys can be modified inplace if needed.

Note that the checks performed when calling *load_state_dict()* with `strict=True` are affected by modifications the hook makes to `missing_keys` or `unexpected_keys`, as expected. Additions to either set of keys will result in an error being thrown when `strict=True`, and clearning out both missing and unexpected keys will avoid an error.

Returns a handle that can be used to remove the added hook by calling `handle.remove()`

Return type `torch.utils.hooks.RemovableHandle`

register_module(*name: str*, *module: Optional[torch.nn.modules.module.Module]*) → None
Alias for *add_module()*.

register_parameter(*name: str*, *param: Optional[torch.nn.parameter.Parameter]*) → None
Adds a parameter to the module.

The parameter can be accessed as an attribute using given name.

Parameters

- **name** (*str*) – name of the parameter. The parameter can be accessed from this module using the given name
- **param** (*Parameter or None*) – parameter to be added to the module. If `None`, then operations that run on parameters, such as *cuda*, are ignored. If `None`, the parameter is **not** included in the module's *state_dict*.

requires_grad_(*requires_grad: bool = True*) → torch.nn.modules.module.T
Change if autograd should record operations on parameters in this module.

This method sets the parameters' `requires_grad` attributes in-place.

This method is helpful for freezing part of the module for finetuning or training parts of a model individually (e.g., GAN training).

See locally-disable-grad-doc for a comparison between *.requires_grad_()* and several similar mechanisms that may be confused with it.

Parameters **requires_grad** (*bool*) – whether autograd should record operations on parameters in this module. Default: `True`.

Returns self

Return type Module

set_extra_state(*state: Any*)
This function is called from *load_state_dict()* to handle any extra state found within the *state_dict*. Implement this function and a corresponding *get_extra_state()* for your module if you need to store extra state within its *state_dict*.

Parameters **state** (*dict*) – Extra state from the *state_dict*

share_memory() → torch.nn.modules.module.T
See `torch.Tensor.share_memory_()`

state_dict(**args*, *destination=None*, *prefix=''*, *keep_vars=False*)
Returns a dictionary containing references to the whole state of the module.

Both parameters and persistent buffers (e.g. running averages) are included. Keys are corresponding parameter and buffer names. Parameters and buffers set to `None` are not included.

Note: The returned object is a shallow copy. It contains references to the module's parameters and buffers.

Warning: Currently `state_dict()` also accepts positional arguments for `destination`, `prefix` and `keep_vars` in order. However, this is being deprecated and keyword arguments will be enforced in future releases.

Warning: Please avoid the use of argument `destination` as it is not designed for end-users.

Parameters

- **destination** (*dict, optional*) – If provided, the state of module will be updated into the dict and the same object is returned. Otherwise, an `OrderedDict` will be created and returned. Default: `None`.
- **prefix** (*str, optional*) – a prefix added to parameter and buffer names to compose the keys in state_dict. Default: `''`.
- **keep_vars** (*bool, optional*) – by default the `Tensor` s returned in the state dict are detached from autograd. If it's set to `True`, detaching will not be performed. Default: `False`.

Returns a dictionary containing a whole state of the module

Return type dict

Example:

```
>>> # xdoctest: +SKIP("undefined vars")
>>> module.state_dict().keys()
['bias', 'weight']
```

to(**args*, ***kwargs*)

Moves and/or casts the parameters and buffers.

This can be called as

to(*device=None*, *dtype=None*, *non_blocking=False*)

to(*dtype*, *non_blocking=False*)

to(*tensor*, *non_blocking=False*)

to(*memory_format=torch.channels_last*)

Its signature is similar to `torch.Tensor.to()`, but only accepts floating point or complex `dtype`s. In addition, this method will only cast the floating point or complex parameters and buffers to `dtype` (if given). The integral parameters and buffers will be moved `device`, if that is given, but with dtypes unchanged. When `non_blocking` is set, it tries to convert/move asynchronously with respect to the host if possible, e.g., moving CPU Tensors with pinned memory to CUDA devices.

See below for examples.

Note: This method modifies the module in-place.

Parameters

- **device** (`torch.device`) – the desired device of the parameters and buffers in this module
- **dtype** (`torch.dtype`) – the desired floating point or complex dtype of the parameters and buffers in this module
- **tensor** (*torch.Tensor*) – Tensor whose dtype and device are the desired dtype and device for all parameters and buffers in this module

- **memory_format** (torch.memory_format) – the desired memory format for 4D parameters and buffers in this module (keyword only argument)

Returns self

Return type Module

Examples:

```
>>> # xdoctest: +IGNORE_WANT("non-deterministic")
>>> linear = nn.Linear(2, 2)
>>> linear.weight
Parameter containing:
tensor([[ 0.1913, -0.3420],
        [-0.5113, -0.2325]])
>>> linear.to(torch.double)
Linear(in_features=2, out_features=2, bias=True)
>>> linear.weight
Parameter containing:
tensor([[ 0.1913, -0.3420],
        [-0.5113, -0.2325]], dtype=torch.float64)
>>> # xdoctest: +REQUIRES(env:TORCH_DOCTEST_CUDA1)
>>> gpu1 = torch.device("cuda:1")
>>> linear.to(gpu1, dtype=torch.half, non_blocking=True)
Linear(in_features=2, out_features=2, bias=True)
>>> linear.weight
Parameter containing:
tensor([[ 0.1914, -0.3420],
        [-0.5112, -0.2324]], dtype=torch.float16, device='cuda:1')
>>> cpu = torch.device("cpu")
>>> linear.to(cpu)
Linear(in_features=2, out_features=2, bias=True)
>>> linear.weight
Parameter containing:
tensor([[ 0.1914, -0.3420],
        [-0.5112, -0.2324]], dtype=torch.float16)

>>> linear = nn.Linear(2, 2, bias=None).to(torch.cdouble)
>>> linear.weight
Parameter containing:
tensor([[ 0.3741+0.j,  0.2382+0.j],
        [ 0.5593+0.j, -0.4443+0.j]], dtype=torch.complex128)
>>> linear(torch.ones(3, 2, dtype=torch.cdouble))
tensor([[0.6122+0.j, 0.1150+0.j],
        [0.6122+0.j, 0.1150+0.j],
        [0.6122+0.j, 0.1150+0.j]], dtype=torch.complex128)
```

to_empty(*, *device: Union[str, torch.device]*) → torch.nn.modules.module.T

Moves the parameters and buffers to the specified device without copying storage.

Parameters **device** (torch.device) – The desired device of the parameters and buffers in this module.

Returns self

Return type Module

train(*mode: bool = True*) → torch.nn.modules.module.T
Sets the module in training mode.

This has any effect only on certain modules. See documentations of particular modules for details of their behaviors in training/evaluation mode, if they are affected, e.g. `Dropout`, `BatchNorm`, etc.

Parameters **mode** (*bool*) – whether to set training mode (`True`) or evaluation mode (`False`). Default: `True`.

Returns self

Return type Module

training: bool

type(*dst_type: Union[torch.dtype, str]*) → torch.nn.modules.module.T
Casts all parameters and buffers to `dst_type`.

Note: This method modifies the module in-place.

Parameters **dst_type** (*type or string*) – the desired type

Returns self

Return type Module

xpu(*device: Optional[Union[int, torch.device]] = None*) → torch.nn.modules.module.T
Moves all model parameters and buffers to the XPU.

This also makes associated parameters and buffers different objects. So it should be called before constructing optimizer if the module will live on XPU while being optimized.

Note: This method modifies the module in-place.

Parameters **device** (*int, optional*) – if specified, all parameters will be copied to that device

Returns self

Return type Module

zero_grad(*set_to_none: bool = False*) → None
Sets gradients of all model parameters to zero. See similar function under `torch.optim.Optimizer` for more context.

Parameters **set_to_none** (*bool*) – instead of setting to zero, set the grads to None. See `torch.optim.Optimizer.zero_grad()` for details.

synthcity.plugins.core.models.tabular_flows module

class TabularFlows(*X: pandas.core.frame.DataFrame*, *n_iter: int = 1000*, *n_layers_hidden: int = 5*, *n_units_hidden: int = 10*, *batch_size: int = 100*, *num_transform_blocks: int = 2*, *dropout: float = 0.25*, *batch_norm: bool = False*, *num_bins: int = 8*, *tail_bound: float = 3*, *lr: float = 0.001*, *apply_unconditional_transform: bool = True*, *base_distribution: str = 'standard_normal'*, *linear_transform_type: str = 'permutation'*, *base_transform_type: str = 'rq-autoregressive'*, *encoder_max_clusters: int = 20*, *encoder_whitelist: list = []*, *device: Any = device(type='cpu')*, *n_iter_min: int = 100*, *n_iter_print: int = 10*, *patience: int = 10*, *patience_metric: Optional[synthcity.metrics.weighted_metrics.WeightedMetrics] = None*)

Bases: `torch.nn.modules.module.Module`

Normalizing flow for tabular data.

This class combines normalizing flow and tabular encoder to form a generative model for tabular data.

Parameters

- **n_iter** – int Number of flow steps
- **n_layers_hidden** – int Number of transformation layers
- **n_units_hidden** – int Number of hidden units for each layer
- **batch_size** – int Size of batch used for training
- **num_transform_blocks** – int Number of blocks to use in coupling/autoregressive nets.
- **dropout** – float Dropout probability for coupling/autoregressive nets.
- **batch_norm** – bool Whether to use batch norm in coupling/autoregressive nets.
- **num_bins** – int Number of bins to use for piecewise transforms.
- **tail_bound** – float Box is on [-bound, bound]^2
- **lr** – float Learning rate for optimizer.
- **apply_unconditional_transform** – bool Whether to unconditionally transform 'identity' features in the coupling layer.
- **base_distribution** – str Possible values: "standard_normal"
- **linear_transform_type** – str Type of linear transform to use. Possible values:
  - lu : A linear transform where we parameterize the LU decomposition of the weights.
  - permutation: Permutes using a random, but fixed, permutation.
  - svd: A linear module using the SVD decomposition for the weight matrix.
- **base_transform_type** – str Type of transform to use between linear layers. Possible values:
  - **affine-coupling** [An affine coupling layer that scales and shifts part of the variables.] Ref: L. Dinh et al., "Density estimation using Real NVP".
  - **quadratic-coupling :** Ref: Müller et al., "Neural Importance Sampling".
  - **rq-coupling** [Rational Quadratic Coupling] Ref: Durkan et al, "Neural Spline Flows".
  - **affine-autoregressive :Affine Autoregressive Transform** Ref: Durkan et al, "Neural Spline Flows".

  - **quadratic-autoregressive** [Quadratic Autoregressive Transform] Ref: Durkan et al, "Neural Spline Flows".
  - **rq-autoregressive** [Rational Quadratic Autoregressive Transform] Ref: Durkan et al, "Neural Spline Flows".
- **stopping** (*# early*) –
- **n_iter_print** – int Number of iterations after which to print updates and check the validation loss.
- **n_iter_min** – int Minimum number of iterations to go through before starting early stopping
- **patience** – int Max number of iterations without any improvement before early stopping is trigged.
- **patience_metric** – WeightedMetrics Metric evaluator

T_destination
alias of TypeVar('T_destination', bound=`Dict[str, Any]`)

add_module(*name: str*, *module: Optional[torch.nn.modules.module.Module]*) → None
Adds a child module to the current module.

The module can be accessed as an attribute using the given name.

Parameters

- **name** (*str*) – name of the child module. The child module can be accessed from this module using the given name
- **module** (*Module*) – child module to be added to the module.

apply(*fn: Callable[[torch.nn.modules.module.Module], None]*) → torch.nn.modules.module.T
Applies `fn` recursively to every submodule (as returned by `.children()`) as well as self. Typical use includes initializing the parameters of a model (see also nn-init-doc).

Parameters **fn** (`Module` -> None) – function to be applied to each submodule

Returns self

Return type Module

Example:

```
>>> @torch.no_grad()
>>> def init_weights(m):
>>>     print(m)
>>>     if type(m) == nn.Linear:
>>>         m.weight.fill_(1.0)
>>>         print(m.weight)
>>> net = nn.Sequential(nn.Linear(2, 2), nn.Linear(2, 2))
>>> net.apply(init_weights)
Linear(in_features=2, out_features=2, bias=True)
Parameter containing:
tensor([[1., 1.],
        [1., 1.]], requires_grad=True)
Linear(in_features=2, out_features=2, bias=True)
Parameter containing:
tensor([[1., 1.],
        [1., 1.]], requires_grad=True)
```

(continues on next page)

(continued from previous page)

```
Sequential(
  (0): Linear(in_features=2, out_features=2, bias=True)
  (1): Linear(in_features=2, out_features=2, bias=True)
)
```

bfloat16() → torch.nn.modules.module.T
Casts all floating point parameters and buffers to `bfloat16` datatype.

Note: This method modifies the module in-place.

Returns self

Return type Module

buffers(*recurse: bool = True*) → Iterator[torch.Tensor]
Returns an iterator over module buffers.

Parameters **recurse** (*bool*) – if True, then yields buffers of this module and all submodules. Otherwise, yields only buffers that are direct members of this module.

Yields *torch.Tensor* – module buffer

Example:

```
>>> # xdoctest: +SKIP("undefined vars")
>>> for buf in model.buffers():
>>>     print(type(buf), buf.size())
<class 'torch.Tensor'> (20L,)
<class 'torch.Tensor'> (20L, 1L, 5L, 5L)
```

children() → Iterator[torch.nn.modules.module.Module]
Returns an iterator over immediate children modules.

Yields *Module* – a child module

cpu() → torch.nn.modules.module.T
Moves all model parameters and buffers to the CPU.

Note: This method modifies the module in-place.

Returns self

Return type Module

cuda(*device: Optional[Union[int, torch.device]] = None*) → torch.nn.modules.module.T
Moves all model parameters and buffers to the GPU.

This also makes associated parameters and buffers different objects. So it should be called before constructing optimizer if the module will live on GPU while being optimized.

Note: This method modifies the module in-place.

Parameters **device** (*int, optional*) – if specified, all parameters will be copied to that device

Returns self

Return type Module

decode(*X: pandas.core.frame.DataFrame*) → pandas.core.frame.DataFrame

double() → torch.nn.modules.module.T
Casts all floating point parameters and buffers to `double` datatype.

Note: This method modifies the module in-place.

Returns self

Return type Module

dump_patches: bool = False

encode(*X: pandas.core.frame.DataFrame*) → pandas.core.frame.DataFrame

eval() → torch.nn.modules.module.T
Sets the module in evaluation mode.

This has any effect only on certain modules. See documentations of particular modules for details of their behaviors in training/evaluation mode, if they are affected, e.g. `Dropout`, `BatchNorm`, etc.

This is equivalent with `self.train(False)`.

See locally-disable-grad-doc for a comparison between *.eval()* and several similar mechanisms that may be confused with it.

Returns self

Return type Module

extra_repr() → str
Set the extra representation of the module

To print customized extra information, you should re-implement this method in your own modules. Both single-line and multi-line strings are acceptable.

fit(*X: pandas.core.frame.DataFrame*) → Any

float() → torch.nn.modules.module.T
Casts all floating point parameters and buffers to `float` datatype.

Note: This method modifies the module in-place.

Returns self

Return type Module

forward(*count: int*) → torch.Tensor
Defines the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

generate(*count: int*) → pandas.core.frame.DataFrame

get_buffer(*target: str*) → torch.Tensor

Returns the buffer given by `target` if it exists, otherwise throws an error.

See the docstring for `get_submodule` for a more detailed explanation of this method's functionality as well as how to correctly specify `target`.

Parameters **target** – The fully-qualified string name of the buffer to look for. (See `get_submodule` for how to specify a fully-qualified string.)

Returns The buffer referenced by `target`

Return type torch.Tensor

Raises **AttributeError** – If the target string references an invalid path or resolves to something that is not a buffer

get_encoder() → *synthcity.plugins.core.models.tabular_encoder.TabularEncoder*

get_extra_state() → Any

Returns any extra state to include in the module's state_dict. Implement this and a corresponding *set_extra_state()* for your module if you need to store extra state. This function is called when building the module's *state_dict()*.

Note that extra state should be pickleable to ensure working serialization of the state_dict. We only provide provide backwards compatibility guarantees for serializing Tensors; other objects may break backwards compatibility if their serialized pickled form changes.

Returns Any extra state to store in the module's state_dict

Return type object

get_parameter(*target: str*) → torch.nn.parameter.Parameter

Returns the parameter given by `target` if it exists, otherwise throws an error.

See the docstring for `get_submodule` for a more detailed explanation of this method's functionality as well as how to correctly specify `target`.

Parameters **target** – The fully-qualified string name of the Parameter to look for. (See `get_submodule` for how to specify a fully-qualified string.)

Returns The Parameter referenced by `target`

Return type torch.nn.Parameter

Raises **AttributeError** – If the target string references an invalid path or resolves to something that is not an `nn.Parameter`

get_submodule(*target: str*) → torch.nn.modules.module.Module

Returns the submodule given by `target` if it exists, otherwise throws an error.

For example, let's say you have an `nn.Module` `A` that looks like this:

```
A(
    (net_b): Module(
        (net_c): Module(
```

(continues on next page)

(continued from previous page)

```
            (conv): Conv2d(16, 33, kernel_size=(3, 3), stride=(2, 2))
        )
        (linear): Linear(in_features=100, out_features=200, bias=True)
    )
)
```

(The diagram shows an `nn.Module` `A`. `A` has a nested submodule `net_b`, which itself has two submodules `net_c` and `linear`. `net_c` then has a submodule `conv`.)

To check whether or not we have the `linear` submodule, we would call `get_submodule("net_b.linear")`. To check whether we have the `conv` submodule, we would call `get_submodule("net_b.net_c.conv")`.

The runtime of `get_submodule` is bounded by the degree of module nesting in `target`. A query against `named_modules` achieves the same result, but it is O(N) in the number of transitive modules. So, for a simple check to see if some submodule exists, `get_submodule` should always be used.

Parameters `target` – The fully-qualified string name of the submodule to look for. (See above example for how to specify a fully-qualified string.)

Returns The submodule referenced by `target`

Return type torch.nn.Module

Raises `AttributeError` – If the target string references an invalid path or resolves to something that is not an `nn.Module`

`half`() → torch.nn.modules.module.T

Casts all floating point parameters and buffers to `half` datatype.

Note: This method modifies the module in-place.

Returns self

Return type Module

`ipu`(*device: Optional[Union[int, torch.device]] = None*) → torch.nn.modules.module.T

Moves all model parameters and buffers to the IPU.

This also makes associated parameters and buffers different objects. So it should be called before constructing optimizer if the module will live on IPU while being optimized.

Note: This method modifies the module in-place.

Parameters `device` (*int, optional*) – if specified, all parameters will be copied to that device

Returns self

Return type Module

`load_state_dict`(*state_dict: Mapping[str, Any]*, *strict: bool = True*)

Copies parameters and buffers from *state_dict* into this module and its descendants. If `strict` is `True`, then the keys of *state_dict* must exactly match the keys returned by this module's `state_dict()` function.

Parameters

- **state_dict** (*dict*) – a dict containing parameters and persistent buffers.
- **strict** (*bool, optional*) – whether to strictly enforce that the keys in *state_dict* match the keys returned by this module's `state_dict()` function. Default: `True`

Returns

- **missing_keys** is a list of str containing the missing keys
- **unexpected_keys** is a list of str containing the unexpected keys

Return type `NamedTuple` with `missing_keys` and `unexpected_keys` fields

Note: If a parameter or buffer is registered as `None` and its corresponding key exists in *state_dict*, *load_state_dict()* will raise a `RuntimeError`.

---

modules() → Iterator[torch.nn.modules.module.Module]

Returns an iterator over all modules in the network.

Yields *Module* – a module in the network

Note: Duplicate modules are returned only once. In the following example, `l` will be returned only once.

Example:

```
>>> l = nn.Linear(2, 2)
>>> net = nn.Sequential(l, l)
>>> for idx, m in enumerate(net.modules()):
...     print(idx, '->', m)

0 -> Sequential(
  (0): Linear(in_features=2, out_features=2, bias=True)
  (1): Linear(in_features=2, out_features=2, bias=True)
)
1 -> Linear(in_features=2, out_features=2, bias=True)
```

named_buffers(*prefix: str = ''*, *recurse: bool = True*) → Iterator[Tuple[str, torch.Tensor]]

Returns an iterator over module buffers, yielding both the name of the buffer as well as the buffer itself.

Parameters

- **prefix** (*str*) – prefix to prepend to all buffer names.
- **recurse** (*bool*) – if True, then yields buffers of this module and all submodules. Otherwise, yields only buffers that are direct members of this module.

Yields *(str, torch.Tensor)* – Tuple containing the name and buffer

Example:

```
>>> # xdoctest: +SKIP("undefined vars")
>>> for name, buf in self.named_buffers():
>>>    if name in ['running_var']:
>>>        print(buf.size())
```

named_children() → Iterator[Tuple[str, torch.nn.modules.module.Module]]
Returns an iterator over immediate children modules, yielding both the name of the module as well as the module itself.

Yields *(str, Module)* – Tuple containing a name and child module

Example:

```
>>> # xdoctest: +SKIP("undefined vars")
>>> for name, module in model.named_children():
>>>     if name in ['conv4', 'conv5']:
>>>         print(module)
```

named_modules(*memo: Optional[Set[torch.nn.modules.module.Module]] = None*, *prefix: str = ''*, *remove_duplicate: bool = True*)
Returns an iterator over all modules in the network, yielding both the name of the module as well as the module itself.

Parameters

- **memo** – a memo to store the set of modules already added to the result
- **prefix** – a prefix that will be added to the name of the module
- **remove_duplicate** – whether to remove the duplicated module instances in the result or not

Yields *(str, Module)* – Tuple of name and module

Note: Duplicate modules are returned only once. In the following example, `l` will be returned only once.

Example:

```
>>> l = nn.Linear(2, 2)
>>> net = nn.Sequential(l, l)
>>> for idx, m in enumerate(net.named_modules()):
...     print(idx, '->', m)

0 -> ('', Sequential(
  (0): Linear(in_features=2, out_features=2, bias=True)
  (1): Linear(in_features=2, out_features=2, bias=True)
))
1 -> ('0', Linear(in_features=2, out_features=2, bias=True))
```

named_parameters(*prefix: str = ''*, *recurse: bool = True*) → Iterator[Tuple[str, torch.nn.parameter.Parameter]]
Returns an iterator over module parameters, yielding both the name of the parameter as well as the parameter itself.

Parameters

- **prefix** (*str*) – prefix to prepend to all parameter names.
- **recurse** (*bool*) – if True, then yields parameters of this module and all submodules. Otherwise, yields only parameters that are direct members of this module.

Yields *(str, Parameter)* – Tuple containing the name and parameter

Example:

```
>>> # xdoctest: +SKIP("undefined vars")
>>> for name, param in self.named_parameters():
>>>    if name in ['bias']:
>>>        print(param.size())
```

parameters(*recurse: bool = True*) → Iterator[torch.nn.parameter.Parameter]

Returns an iterator over module parameters.

This is typically passed to an optimizer.

Parameters **recurse** (*bool*) – if True, then yields parameters of this module and all submodules. Otherwise, yields only parameters that are direct members of this module.

Yields *Parameter* – module parameter

Example:

```
>>> # xdoctest: +SKIP("undefined vars")
>>> for param in model.parameters():
>>>     print(type(param), param.size())
<class 'torch.Tensor'> (20L,)
<class 'torch.Tensor'> (20L, 1L, 5L, 5L)
```

register_backward_hook(*hook: Callable[[torch.nn.modules.module.Module, Union[Tuple[torch.Tensor, ...], torch.Tensor], Union[Tuple[torch.Tensor, ...], torch.Tensor]], Union[None, torch.Tensor]]*) → torch.utils.hooks.RemovableHandle

Registers a backward hook on the module.

This function is deprecated in favor of `register_full_backward_hook()` and the behavior of this function will change in future versions.

Returns a handle that can be used to remove the added hook by calling `handle.remove()`

Return type `torch.utils.hooks.RemovableHandle`

register_buffer(*name: str*, *tensor: Optional[torch.Tensor]*, *persistent: bool = True*) → None

Adds a buffer to the module.

This is typically used to register a buffer that should not to be considered a model parameter. For example, BatchNorm's `running_mean` is not a parameter, but is part of the module's state. Buffers, by default, are persistent and will be saved alongside parameters. This behavior can be changed by setting `persistent` to `False`. The only difference between a persistent buffer and a non-persistent buffer is that the latter will not be a part of this module's *state_dict*.

Buffers can be accessed as attributes using given names.

Parameters

- **name** (*str*) – name of the buffer. The buffer can be accessed from this module using the given name
- **tensor** (*Tensor or None*) – buffer to be registered. If `None`, then operations that run on buffers, such as *cuda*, are ignored. If `None`, the buffer is **not** included in the module's *state_dict*.
- **persistent** (*bool*) – whether the buffer is part of this module's *state_dict*.

Example:

```
>>> # xdoctest: +SKIP("undefined vars")
>>> self.register_buffer('running_mean', torch.zeros(num_features))
```

register_forward_hook(*hook: Callable[[...], None]*) → torch.utils.hooks.RemovableHandle
Registers a forward hook on the module.

The hook will be called every time after *forward()* has computed an output. It should have the following signature:

```
hook(module, input, output) -> None or modified output
```

The input contains only the positional arguments given to the module. Keyword arguments won't be passed to the hooks and only to the `forward`. The hook can modify the output. It can modify the input inplace but it will not have effect on forward since this is called after *forward()* is called.

Returns a handle that can be used to remove the added hook by calling `handle.remove()`

Return type `torch.utils.hooks.RemovableHandle`

register_forward_pre_hook(*hook: Callable[[...], None]*) → torch.utils.hooks.RemovableHandle
Registers a forward pre-hook on the module.

The hook will be called every time before *forward()* is invoked. It should have the following signature:

```
hook(module, input) -> None or modified input
```

The input contains only the positional arguments given to the module. Keyword arguments won't be passed to the hooks and only to the `forward`. The hook can modify the input. User can either return a tuple or a single modified value in the hook. We will wrap the value into a tuple if a single value is returned(unless that value is already a tuple).

Returns a handle that can be used to remove the added hook by calling `handle.remove()`

Return type `torch.utils.hooks.RemovableHandle`

register_full_backward_hook(*hook: Callable[[torch.nn.modules.module.Module, Union[Tuple[torch.Tensor, ...], torch.Tensor], Union[Tuple[torch.Tensor, ...], torch.Tensor]], Union[None, torch.Tensor]]*) → torch.utils.hooks.RemovableHandle
Registers a backward hook on the module.

The hook will be called every time the gradients with respect to a module are computed, i.e. the hook will execute if and only if the gradients with respect to module outputs are computed. The hook should have the following signature:

```
hook(module, grad_input, grad_output) -> tuple(Tensor) or None
```

The `grad_input` and `grad_output` are tuples that contain the gradients with respect to the inputs and outputs respectively. The hook should not modify its arguments, but it can optionally return a new gradient with respect to the input that will be used in place of `grad_input` in subsequent computations. `grad_input` will only correspond to the inputs given as positional arguments and all kwarg arguments are ignored. Entries in `grad_input` and `grad_output` will be `None` for all non-Tensor arguments.

For technical reasons, when this hook is applied to a Module, its forward function will receive a view of each Tensor passed to the Module. Similarly the caller will receive a view of each Tensor returned by the Module's forward function.

Warning: Modifying inputs or outputs inplace is not allowed when using backward hooks and will raise an error.

Returns a handle that can be used to remove the added hook by calling `handle.remove()`

Return type torch.utils.hooks.RemovableHandle

register_load_state_dict_post_hook(*hook*)
Registers a post hook to be run after module's load_state_dict is called.

It should have the following signature:: hook(module, incompatible_keys) -> None

The module argument is the current module that this hook is registered on, and the incompatible_keys argument is a NamedTuple consisting of attributes missing_keys and unexpected_keys. missing_keys is a list of str containing the missing keys and unexpected_keys is a list of str containing the unexpected keys.

The given incompatible_keys can be modified inplace if needed.

Note that the checks performed when calling *load_state_dict()* with strict=True are affected by modifications the hook makes to missing_keys or unexpected_keys, as expected. Additions to either set of keys will result in an error being thrown when strict=True, and clearning out both missing and unexpected keys will avoid an error.

Returns a handle that can be used to remove the added hook by calling handle.remove()

Return type torch.utils.hooks.RemovableHandle

register_module(*name: str*, *module: Optional[torch.nn.modules.module.Module]*) → None
Alias for *add_module()*.

register_parameter(*name: str*, *param: Optional[torch.nn.parameter.Parameter]*) → None
Adds a parameter to the module.

The parameter can be accessed as an attribute using given name.

Parameters

- **name** (*str*) – name of the parameter. The parameter can be accessed from this module using the given name
- **param** (*Parameter or None*) – parameter to be added to the module. If None, then operations that run on parameters, such as *cuda*, are ignored. If None, the parameter is **not** included in the module's *state_dict*.

requires_grad_(*requires_grad: bool = True*) → torch.nn.modules.module.T
Change if autograd should record operations on parameters in this module.

This method sets the parameters' requires_grad attributes in-place.

This method is helpful for freezing part of the module for finetuning or training parts of a model individually (e.g., GAN training).

See locally-disable-grad-doc for a comparison between *.requires_grad_()* and several similar mechanisms that may be confused with it.

Parameters **requires_grad** (*bool*) – whether autograd should record operations on parameters in this module. Default: True.

Returns self

Return type Module

set_extra_state(*state: Any*)
This function is called from *load_state_dict()* to handle any extra state found within the *state_dict*. Implement this function and a corresponding *get_extra_state()* for your module if you need to store extra state within its *state_dict*.

Parameters **state** (*dict*) – Extra state from the *state_dict*

share_memory() → torch.nn.modules.module.T
See `torch.Tensor.share_memory_()`

state_dict(**args*, *destination=None*, *prefix=''*, *keep_vars=False*)
Returns a dictionary containing references to the whole state of the module.

Both parameters and persistent buffers (e.g. running averages) are included. Keys are corresponding parameter and buffer names. Parameters and buffers set to `None` are not included.

Note: The returned object is a shallow copy. It contains references to the module's parameters and buffers.

Warning: Currently `state_dict()` also accepts positional arguments for `destination`, `prefix` and `keep_vars` in order. However, this is being deprecated and keyword arguments will be enforced in future releases.

Warning: Please avoid the use of argument `destination` as it is not designed for end-users.

Parameters

- **destination** (*dict, optional*) – If provided, the state of module will be updated into the dict and the same object is returned. Otherwise, an `OrderedDict` will be created and returned. Default: `None`.
- **prefix** (*str, optional*) – a prefix added to parameter and buffer names to compose the keys in state_dict. Default: `''`.
- **keep_vars** (*bool, optional*) – by default the `Tensor` s returned in the state dict are detached from autograd. If it's set to `True`, detaching will not be performed. Default: `False`.

Returns a dictionary containing a whole state of the module

Return type dict

Example:

```
>>> # xdoctest: +SKIP("undefined vars")
>>> module.state_dict().keys()
['bias', 'weight']
```

to(**args*, ***kwargs*)
Moves and/or casts the parameters and buffers.

This can be called as

to(*device=None*, *dtype=None*, *non_blocking=False*)

to(*dtype*, *non_blocking=False*)

to(*tensor*, *non_blocking=False*)

to(*memory_format=torch.channels_last*)

Its signature is similar to `torch.Tensor.to()`, but only accepts floating point or complex `dtypes`. In addition, this method will only cast the floating point or complex parameters and buffers to `dtype` (if given). The integral parameters and buffers will be moved `device`, if that is given, but with dtypes unchanged.

When non_blocking is set, it tries to convert/move asynchronously with respect to the host if possible, e.g., moving CPU Tensors with pinned memory to CUDA devices.

See below for examples.

Note: This method modifies the module in-place.

Parameters

- **device** (torch.device) – the desired device of the parameters and buffers in this module
- **dtype** (torch.dtype) – the desired floating point or complex dtype of the parameters and buffers in this module
- **tensor** (*torch.Tensor*) – Tensor whose dtype and device are the desired dtype and device for all parameters and buffers in this module
- **memory_format** (torch.memory_format) – the desired memory format for 4D parameters and buffers in this module (keyword only argument)

Returns self

Return type Module

Examples:

```
>>> # xdoctest: +IGNORE_WANT("non-deterministic")
>>> linear = nn.Linear(2, 2)
>>> linear.weight
Parameter containing:
tensor([[ 0.1913, -0.3420],
        [-0.5113, -0.2325]])
>>> linear.to(torch.double)
Linear(in_features=2, out_features=2, bias=True)
>>> linear.weight
Parameter containing:
tensor([[ 0.1913, -0.3420],
        [-0.5113, -0.2325]], dtype=torch.float64)
>>> # xdoctest: +REQUIRES(env:TORCH_DOCTEST_CUDA1)
>>> gpu1 = torch.device("cuda:1")
>>> linear.to(gpu1, dtype=torch.half, non_blocking=True)
Linear(in_features=2, out_features=2, bias=True)
>>> linear.weight
Parameter containing:
tensor([[ 0.1914, -0.3420],
        [-0.5112, -0.2324]], dtype=torch.float16, device='cuda:1')
>>> cpu = torch.device("cpu")
>>> linear.to(cpu)
Linear(in_features=2, out_features=2, bias=True)
>>> linear.weight
Parameter containing:
tensor([[ 0.1914, -0.3420],
        [-0.5112, -0.2324]], dtype=torch.float16)
```

(continues on next page)

(continued from previous page)

```
>>> linear = nn.Linear(2, 2, bias=None).to(torch.cdouble)
>>> linear.weight
Parameter containing:
tensor([[ 0.3741+0.j,  0.2382+0.j],
        [ 0.5593+0.j, -0.4443+0.j]], dtype=torch.complex128)
>>> linear(torch.ones(3, 2, dtype=torch.cdouble))
tensor([[0.6122+0.j, 0.1150+0.j],
        [0.6122+0.j, 0.1150+0.j],
        [0.6122+0.j, 0.1150+0.j]], dtype=torch.complex128)
```

to_empty(*, *device: Union[str, torch.device]*) → torch.nn.modules.module.T
Moves the parameters and buffers to the specified device without copying storage.

Parameters **device** (torch.device) – The desired device of the parameters and buffers in this module.

Returns self

Return type Module

train(*mode: bool = True*) → torch.nn.modules.module.T
Sets the module in training mode.

This has any effect only on certain modules. See documentations of particular modules for details of their behaviors in training/evaluation mode, if they are affected, e.g. Dropout, BatchNorm, etc.

Parameters **mode** (*bool*) – whether to set training mode (True) or evaluation mode (False). Default: True.

Returns self

Return type Module

training: bool

type(*dst_type: Union[torch.dtype, str]*) → torch.nn.modules.module.T
Casts all parameters and buffers to dst_type.

Note: This method modifies the module in-place.

Parameters **dst_type** (*type or string*) – the desired type

Returns self

Return type Module

xpu(*device: Optional[Union[int, torch.device]] = None*) → torch.nn.modules.module.T
Moves all model parameters and buffers to the XPU.

This also makes associated parameters and buffers different objects. So it should be called before constructing optimizer if the module will live on XPU while being optimized.

Note: This method modifies the module in-place.

Parameters **device** (*int, optional*) – if specified, all parameters will be copied to that device

Returns self

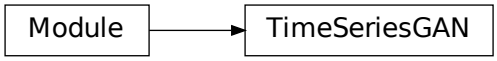
Return type Module

`zero_grad`(*set_to_none: bool = False*) → None
Sets gradients of all model parameters to zero. See similar function under `torch.optim.Optimizer` for more context.

Parameters `set_to_none` (*bool*) – instead of setting to zero, set the grads to None. See `torch.optim.Optimizer.zero_grad()` for details.

synthcity.plugins.core.models.ts_gan module

`class TimeSeriesGAN`(*n_static_units: int*, *n_static_units_latent: int*, *n_temporal_units: int*, *n_temporal_window: int*, *n_temporal_units_latent: int*, *n_units_conditional: int = 0*, *generator_n_layers_hidden: int = 1*, *generator_n_units_hidden: int = 250*, *generator_nonlin: str = 'leaky_relu'*, *generator_static_nonlin_out: Optional[List[Tuple[str, int]]] = None*, *generator_temporal_nonlin_out: Optional[List[Tuple[str, int]]] = None*, *generator_n_iter: int = 1000*, *generator_batch_norm: bool = False*, *generator_dropout: float = 0*, *generator_loss: Optional[Callable] = None*, *generator_lr: float = 0.0002*, *generator_weight_decay: float = 0.001*, *generator_residual: bool = True*, *discriminator_n_layers_hidden: int = 1*, *discriminator_n_units_hidden: int = 300*, *discriminator_nonlin: str = 'leaky_relu'*, *discriminator_n_iter: int = 1*, *discriminator_batch_norm: bool = False*, *discriminator_dropout: float = 0.1*, *discriminator_loss: Optional[Callable] = None*, *discriminator_lr: float = 0.0002*, *discriminator_weight_decay: float = 0.001*, *batch_size: int = 64*, *n_iter_print: int = 10*, *random_state: int = 0*, *clipping_value: int = 1*, *gamma_penalty: float = 1*, *moments_penalty: float = 100*, *embedding_penalty: float = 10*, *dataloader_sampler: Optional[torch.utils.data.sampler.Sampler] = None*, *mode: str = 'RNN'*, *device: Any = device(type='cpu')*, *use_horizon_condition: bool = True*)
Bases: `torch.nn.modules.module.Module`



Basic TimeSeriesGAN implementation.

Parameters

- `n_static_units` – int, Number of units for the static features
- `n_static_units_latent` – int, Number of latent units for the static features
- `n_temporal_units` – int, Number of units for the temporal features
- `n_temporal_window` – int, Number of temporal observations for each subject
- `n_temporal_units_latent` – int, Number of temporal latent units
- `n_units_conditional` – int = 0, Number of conditional units
- `generator_n_layers_hidden` – int. Default: 1 Number of hidden layers in the generator

- **generator_n_units_hidden** – int. Default: 250 Number of hidden units in each layer of the Generator
- **generator_nonlin** – string, default 'elu' Nonlinearity to use in the generator. Can be 'elu', 'relu', 'selu' or 'leaky_relu'.
- **generator_n_iter** – int. Default: 1000 Maximum number of iterations in the Generator.
- **generator_batch_norm** – bool. Default: False Enable/disable batch norm for the generator
- **generator_dropout** – float. Default: 0 Generator Dropout value. If 0, the dropout is not used.
- **generator_residual** – bool. Default: True Use residuals for the generator
- **generator_lr** – float. Default: 2e-4 Generator learning rate.
- **generator_weight_decay** – float. Default = 1e-3 l2 (ridge) penalty for the generator weights.
- **discriminator_n_layers_hidden** – int. Default = 1 Number of hidden layers in the discriminator
- **discriminator_n_units_hidden** – int. Default = 300 Number of hidden units in each layer of the discriminator
- **discriminator_nonlin** – string, default = 'relu' Nonlinearity to use in the discriminator. Can be 'elu', 'relu', 'selu' or 'leaky_relu'.
- **discriminator_n_iter** – int Maximum number of iterations in the discriminator.
- **discriminator_batch_norm** – bool. Default: False Enable/disable batch norm for the discriminator
- **discriminator_dropout** – float. Default = 0.1 Dropout value for the discriminator. If 0, the dropout is not used.
- **discriminator_lr** – float. Default = 2e-4 learning rate for discriminator optimizer.
- **discriminator_weight_decay** – float. Default = 1e-3 l2 (ridge) penalty for the discriminator weights.
- **batch_size** – int. Default = 64 Batch size
- **n_iter_print** – int Number of iterations after which to print updates and check the validation loss.
- **random_state** – int random_state used
- **clipping_value** – int, default = 0 Gradients clipping value. Zero disables the feature
- **gamma_penalty** – float. Default = 1. Latent representation penalty
- **moments_penalty** – float. Default = 100. Generator Moments(var and mean) penalty
- **embedding_penalty** – float. Default = 10 Embedding representation penalty
- **dataloader_sampler** – Optional[sampler.Sampler] = None Optional data sampler
- **mode** – str = "RNN" Core neural net architecture. Available models:
  - "LSTM"
  - "GRU"

- "RNN"
- "Transformer"
- "MLSTM_FCN"
- "TCN"
- "InceptionTime"
- "InceptionTimePlus"
- "XceptionTime"
- "ResCNN"
- "OmniScaleCNN"
- "XCM"

- **device** – The device used by PyTorch. cpu/cuda
- **use_horizon_condition** – bool. Default = True Whether to condition the covariate generation on the observation times or not.

T_destination
alias of TypeVar('T_destination', bound=Dict[str, Any])

add_module(*name: str*, *module: Optional[torch.nn.modules.module.Module]*) → None
Adds a child module to the current module.

The module can be accessed as an attribute using the given name.

Parameters

- **name** (*str*) – name of the child module. The child module can be accessed from this module using the given name
- **module** (*Module*) – child module to be added to the module.

apply(*fn: Callable[[torch.nn.modules.module.Module], None]*) → torch.nn.modules.module.T
Applies `fn` recursively to every submodule (as returned by `.children()`) as well as self. Typical use includes initializing the parameters of a model (see also nn-init-doc).

Parameters **fn** (`Module` -> None) – function to be applied to each submodule

Returns self

Return type Module

Example:

```
>>> @torch.no_grad()
>>> def init_weights(m):
>>>     print(m)
>>>     if type(m) == nn.Linear:
>>>         m.weight.fill_(1.0)
>>>         print(m.weight)
>>> net = nn.Sequential(nn.Linear(2, 2), nn.Linear(2, 2))
>>> net.apply(init_weights)
Linear(in_features=2, out_features=2, bias=True)
Parameter containing:
tensor([[1., 1.],
        [1., 1.]], requires_grad=True)
```

(continues on next page)

(continued from previous page)

```
Linear(in_features=2, out_features=2, bias=True)
Parameter containing:
tensor([[1., 1.],
        [1., 1.]], requires_grad=True)
Sequential(
  (0): Linear(in_features=2, out_features=2, bias=True)
  (1): Linear(in_features=2, out_features=2, bias=True)
)
```

bfloat16() → torch.nn.modules.module.T
Casts all floating point parameters and buffers to `bfloat16` datatype.

Note: This method modifies the module in-place.

Returns self

Return type Module

buffers(*recurse: bool = True*) → Iterator[torch.Tensor]
Returns an iterator over module buffers.

Parameters **recurse** (*bool*) – if True, then yields buffers of this module and all submodules. Otherwise, yields only buffers that are direct members of this module.

Yields *torch.Tensor* – module buffer

Example:

```
>>> # xdoctest: +SKIP("undefined vars")
>>> for buf in model.buffers():
>>>     print(type(buf), buf.size())
<class 'torch.Tensor'> (20L,)
<class 'torch.Tensor'> (20L, 1L, 5L, 5L)
```

children() → Iterator[torch.nn.modules.module.Module]
Returns an iterator over immediate children modules.

Yields *Module* – a child module

cpu() → torch.nn.modules.module.T
Moves all model parameters and buffers to the CPU.

Note: This method modifies the module in-place.

Returns self

Return type Module

cuda(*device: Optional[Union[int, torch.device]] = None*) → torch.nn.modules.module.T
Moves all model parameters and buffers to the GPU.

This also makes associated parameters and buffers different objects. So it should be called before constructing optimizer if the module will live on GPU while being optimized.

Note: This method modifies the module in-place.

Parameters **device** (*int, optional*) – if specified, all parameters will be copied to that device

Returns self

Return type Module

dataloader(*static_data: torch.Tensor*, *temporal_data: torch.Tensor*, *observation_times: torch.Tensor*, *cond: Optional[torch.Tensor] = None*) → torch.utils.data.dataloader.DataLoader

double() → torch.nn.modules.module.T
Casts all floating point parameters and buffers to `double` datatype.

Note: This method modifies the module in-place.

Returns self

Return type Module

dump_patches: bool = False

eval() → torch.nn.modules.module.T
Sets the module in evaluation mode.

This has any effect only on certain modules. See documentations of particular modules for details of their behaviors in training/evaluation mode, if they are affected, e.g. `Dropout`, `BatchNorm`, etc.

This is equivalent with `self.train(False)`.

See locally-disable-grad-doc for a comparison between *.eval()* and several similar mechanisms that may be confused with it.

Returns self

Return type Module

extra_repr() → str
Set the extra representation of the module

To print customized extra information, you should re-implement this method in your own modules. Both single-line and multi-line strings are acceptable.

fit(*static_data: numpy.ndarray*, *temporal_data: numpy.ndarray*, *observation_times: numpy.ndarray*, *cond: Optional[numpy.ndarray] = None*) → *synthcity.plugins.core.models.ts_gan.TimeSeriesGAN*

float() → torch.nn.modules.module.T
Casts all floating point parameters and buffers to `float` datatype.

Note: This method modifies the module in-place.

Returns self

Return type Module

forward(*count: int*, *cond: Optional[torch.Tensor] = None*, *static_data: Optional[torch.Tensor] = None*, *observation_times: Optional[torch.Tensor] = None*) → Tuple[torch.Tensor, torch.Tensor, torch.Tensor]

Defines the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

---

generate(*count: int*, *cond: Optional[numpy.ndarray] = None*, *static_data: Optional[numpy.ndarray] = None*, *observation_times: Optional[numpy.ndarray] = None*) → Tuple[numpy.ndarray, numpy.ndarray, numpy.ndarray]

get_buffer(*target: str*) → torch.Tensor

Returns the buffer given by `target` if it exists, otherwise throws an error.

See the docstring for `get_submodule` for a more detailed explanation of this method's functionality as well as how to correctly specify `target`.

Parameters **target** – The fully-qualified string name of the buffer to look for. (See `get_submodule` for how to specify a fully-qualified string.)

Returns The buffer referenced by `target`

Return type torch.Tensor

Raises **AttributeError** – If the target string references an invalid path or resolves to something that is not a buffer

get_extra_state() → Any

Returns any extra state to include in the module's state_dict. Implement this and a corresponding *set_extra_state()* for your module if you need to store extra state. This function is called when building the module's *state_dict()*.

Note that extra state should be pickleable to ensure working serialization of the state_dict. We only provide provide backwards compatibility guarantees for serializing Tensors; other objects may break backwards compatibility if their serialized pickled form changes.

Returns Any extra state to store in the module's state_dict

Return type object

get_parameter(*target: str*) → torch.nn.parameter.Parameter

Returns the parameter given by `target` if it exists, otherwise throws an error.

See the docstring for `get_submodule` for a more detailed explanation of this method's functionality as well as how to correctly specify `target`.

Parameters **target** – The fully-qualified string name of the Parameter to look for. (See `get_submodule` for how to specify a fully-qualified string.)

Returns The Parameter referenced by `target`

Return type torch.nn.Parameter

Raises **AttributeError** – If the target string references an invalid path or resolves to something that is not an `nn.Parameter`

get_submodule(*target: str*) → torch.nn.modules.module.Module

Returns the submodule given by `target` if it exists, otherwise throws an error.

For example, let's say you have an nn.Module A that looks like this:

```
A(
    (net_b): Module(
        (net_c): Module(
            (conv): Conv2d(16, 33, kernel_size=(3, 3), stride=(2, 2))
        )
        (linear): Linear(in_features=100, out_features=200, bias=True)
    )
)
```

(The diagram shows an nn.Module A. A has a nested submodule net_b, which itself has two submodules net_c and linear. net_c then has a submodule conv.)

To check whether or not we have the linear submodule, we would call get_submodule("net_b.linear"). To check whether we have the conv submodule, we would call get_submodule("net_b.net_c.conv").

The runtime of get_submodule is bounded by the degree of module nesting in target. A query against named_modules achieves the same result, but it is O(N) in the number of transitive modules. So, for a simple check to see if some submodule exists, get_submodule should always be used.

Parameters **target** – The fully-qualified string name of the submodule to look for. (See above example for how to specify a fully-qualified string.)

Returns The submodule referenced by target

Return type torch.nn.Module

Raises **AttributeError** – If the target string references an invalid path or resolves to something that is not an nn.Module

half() → torch.nn.modules.module.T

Casts all floating point parameters and buffers to half datatype.

Note: This method modifies the module in-place.

Returns self

Return type Module

ipu(*device: Optional[Union[int, torch.device]] = None*) → torch.nn.modules.module.T

Moves all model parameters and buffers to the IPU.

This also makes associated parameters and buffers different objects. So it should be called before constructing optimizer if the module will live on IPU while being optimized.

Note: This method modifies the module in-place.

Parameters **device** (*int, optional*) – if specified, all parameters will be copied to that device

Returns self

Return type Module

load_state_dict(*state_dict: Mapping[str, Any]*, *strict: bool = True*)

Copies parameters and buffers from *state_dict* into this module and its descendants. If `strict` is `True`, then the keys of *state_dict* must exactly match the keys returned by this module's `state_dict()` function.

Parameters

- **state_dict** (*dict*) – a dict containing parameters and persistent buffers.
- **strict** (*bool, optional*) – whether to strictly enforce that the keys in *state_dict* match the keys returned by this module's `state_dict()` function. Default: `True`

Returns

- **missing_keys** is a list of str containing the missing keys
- **unexpected_keys** is a list of str containing the unexpected keys

Return type `NamedTuple` with `missing_keys` and `unexpected_keys` fields

Note: If a parameter or buffer is registered as `None` and its corresponding key exists in *state_dict*, *load_state_dict()* will raise a `RuntimeError`.

---

modules() → Iterator[torch.nn.modules.module.Module]

Returns an iterator over all modules in the network.

Yields *Module* – a module in the network

Note: Duplicate modules are returned only once. In the following example, `l` will be returned only once.

Example:

```
>>> l = nn.Linear(2, 2)
>>> net = nn.Sequential(l, l)
>>> for idx, m in enumerate(net.modules()):
...     print(idx, '->', m)

0 -> Sequential(
  (0): Linear(in_features=2, out_features=2, bias=True)
  (1): Linear(in_features=2, out_features=2, bias=True)
)
1 -> Linear(in_features=2, out_features=2, bias=True)
```

named_buffers(*prefix: str = ''*, *recurse: bool = True*) → Iterator[Tuple[str, torch.Tensor]]

Returns an iterator over module buffers, yielding both the name of the buffer as well as the buffer itself.

Parameters

- **prefix** (*str*) – prefix to prepend to all buffer names.
- **recurse** (*bool*) – if True, then yields buffers of this module and all submodules. Otherwise, yields only buffers that are direct members of this module.

Yields *(str, torch.Tensor)* – Tuple containing the name and buffer

Example:

```
>>> # xdoctest: +SKIP("undefined vars")
>>> for name, buf in self.named_buffers():
>>>    if name in ['running_var']:
>>>        print(buf.size())
```

named_children() → Iterator[Tuple[str, torch.nn.modules.module.Module]]

Returns an iterator over immediate children modules, yielding both the name of the module as well as the module itself.

Yields *(str, Module)* – Tuple containing a name and child module

Example:

```
>>> # xdoctest: +SKIP("undefined vars")
>>> for name, module in model.named_children():
>>>     if name in ['conv4', 'conv5']:
>>>         print(module)
```

named_modules(*memo: Optional[Set[torch.nn.modules.module.Module]] = None*, *prefix: str = ''*, *remove_duplicate: bool = True*)

Returns an iterator over all modules in the network, yielding both the name of the module as well as the module itself.

Parameters

- **memo** – a memo to store the set of modules already added to the result
- **prefix** – a prefix that will be added to the name of the module
- **remove_duplicate** – whether to remove the duplicated module instances in the result or not

Yields *(str, Module)* – Tuple of name and module

Note: Duplicate modules are returned only once. In the following example, `l` will be returned only once.

Example:

```
>>> l = nn.Linear(2, 2)
>>> net = nn.Sequential(l, l)
>>> for idx, m in enumerate(net.named_modules()):
...     print(idx, '->', m)

0 -> ('', Sequential(
  (0): Linear(in_features=2, out_features=2, bias=True)
  (1): Linear(in_features=2, out_features=2, bias=True)
))
1 -> ('0', Linear(in_features=2, out_features=2, bias=True))
```

named_parameters(*prefix: str = ''*, *recurse: bool = True*) → Iterator[Tuple[str, torch.nn.parameter.Parameter]]

Returns an iterator over module parameters, yielding both the name of the parameter as well as the parameter itself.

Parameters

- **prefix** (*str*) – prefix to prepend to all parameter names.

- **recurse** (*bool*) – if True, then yields parameters of this module and all submodules. Otherwise, yields only parameters that are direct members of this module.

Yields *(str, Parameter)* – Tuple containing the name and parameter

Example:

```
>>> # xdoctest: +SKIP("undefined vars")
>>> for name, param in self.named_parameters():
>>>    if name in ['bias']:
>>>        print(param.size())
```

parameters(*recurse: bool = True*) → Iterator[torch.nn.parameter.Parameter]

Returns an iterator over module parameters.

This is typically passed to an optimizer.

Parameters **recurse** (*bool*) – if True, then yields parameters of this module and all submodules. Otherwise, yields only parameters that are direct members of this module.

Yields *Parameter* – module parameter

Example:

```
>>> # xdoctest: +SKIP("undefined vars")
>>> for param in model.parameters():
>>>     print(type(param), param.size())
<class 'torch.Tensor'> (20L,)
<class 'torch.Tensor'> (20L, 1L, 5L, 5L)
```

register_backward_hook(*hook: Callable[[torch.nn.modules.module.Module, Union[Tuple[torch.Tensor, ...], torch.Tensor], Union[Tuple[torch.Tensor, ...], torch.Tensor]], Union[None, torch.Tensor]]*) → torch.utils.hooks.RemovableHandle

Registers a backward hook on the module.

This function is deprecated in favor of `register_full_backward_hook()` and the behavior of this function will change in future versions.

Returns a handle that can be used to remove the added hook by calling `handle.remove()`

Return type `torch.utils.hooks.RemovableHandle`

register_buffer(*name: str*, *tensor: Optional[torch.Tensor]*, *persistent: bool = True*) → None

Adds a buffer to the module.

This is typically used to register a buffer that should not to be considered a model parameter. For example, BatchNorm's `running_mean` is not a parameter, but is part of the module's state. Buffers, by default, are persistent and will be saved alongside parameters. This behavior can be changed by setting `persistent` to `False`. The only difference between a persistent buffer and a non-persistent buffer is that the latter will not be a part of this module's *state_dict*.

Buffers can be accessed as attributes using given names.

Parameters

- **name** (*str*) – name of the buffer. The buffer can be accessed from this module using the given name
- **tensor** (*Tensor or None*) – buffer to be registered. If `None`, then operations that run on buffers, such as *cuda*, are ignored. If `None`, the buffer is **not** included in the module's *state_dict*.

- **persistent** (*bool*) – whether the buffer is part of this module's *state_dict*.

Example:

```
>>> # xdoctest: +SKIP("undefined vars")
>>> self.register_buffer('running_mean', torch.zeros(num_features))
```

register_forward_hook(*hook: Callable[[...], None]*) → torch.utils.hooks.RemovableHandle

Registers a forward hook on the module.

The hook will be called every time after *forward()* has computed an output. It should have the following signature:

```
hook(module, input, output) -> None or modified output
```

The input contains only the positional arguments given to the module. Keyword arguments won't be passed to the hooks and only to the `forward`. The hook can modify the output. It can modify the input inplace but it will not have effect on forward since this is called after *forward()* is called.

Returns a handle that can be used to remove the added hook by calling `handle.remove()`

Return type `torch.utils.hooks.RemovableHandle`

register_forward_pre_hook(*hook: Callable[[...], None]*) → torch.utils.hooks.RemovableHandle

Registers a forward pre-hook on the module.

The hook will be called every time before *forward()* is invoked. It should have the following signature:

```
hook(module, input) -> None or modified input
```

The input contains only the positional arguments given to the module. Keyword arguments won't be passed to the hooks and only to the `forward`. The hook can modify the input. User can either return a tuple or a single modified value in the hook. We will wrap the value into a tuple if a single value is returned(unless that value is already a tuple).

Returns a handle that can be used to remove the added hook by calling `handle.remove()`

Return type `torch.utils.hooks.RemovableHandle`

register_full_backward_hook(*hook: Callable[[torch.nn.modules.module.Module, Union[Tuple[torch.Tensor, ...], torch.Tensor], Union[Tuple[torch.Tensor, ...], torch.Tensor]], Union[None, torch.Tensor]]*) → torch.utils.hooks.RemovableHandle

Registers a backward hook on the module.

The hook will be called every time the gradients with respect to a module are computed, i.e. the hook will execute if and only if the gradients with respect to module outputs are computed. The hook should have the following signature:

```
hook(module, grad_input, grad_output) -> tuple(Tensor) or None
```

The `grad_input` and `grad_output` are tuples that contain the gradients with respect to the inputs and outputs respectively. The hook should not modify its arguments, but it can optionally return a new gradient with respect to the input that will be used in place of `grad_input` in subsequent computations. `grad_input` will only correspond to the inputs given as positional arguments and all kwarg arguments are ignored. Entries in `grad_input` and `grad_output` will be `None` for all non-Tensor arguments.

For technical reasons, when this hook is applied to a Module, its forward function will receive a view of each Tensor passed to the Module. Similarly the caller will receive a view of each Tensor returned by the Module's forward function.

> **Warning:** Modifying inputs or outputs inplace is not allowed when using backward hooks and will raise an error.

Returns a handle that can be used to remove the added hook by calling `handle.remove()`

Return type `torch.utils.hooks.RemovableHandle`

register_load_state_dict_post_hook(*hook*)

Registers a post hook to be run after module's `load_state_dict` is called.

It should have the following signature:: hook(module, incompatible_keys) -> None

The `module` argument is the current module that this hook is registered on, and the `incompatible_keys` argument is a `NamedTuple` consisting of attributes `missing_keys` and `unexpected_keys`. `missing_keys` is a `list` of `str` containing the missing keys and `unexpected_keys` is a `list` of `str` containing the unexpected keys.

The given incompatible_keys can be modified inplace if needed.

Note that the checks performed when calling *load_state_dict()* with `strict=True` are affected by modifications the hook makes to `missing_keys` or `unexpected_keys`, as expected. Additions to either set of keys will result in an error being thrown when `strict=True`, and clearning out both missing and unexpected keys will avoid an error.

Returns a handle that can be used to remove the added hook by calling `handle.remove()`

Return type `torch.utils.hooks.RemovableHandle`

register_module(*name: str*, *module: Optional[torch.nn.modules.module.Module]*) → None

Alias for *add_module()*.

register_parameter(*name: str*, *param: Optional[torch.nn.parameter.Parameter]*) → None

Adds a parameter to the module.

The parameter can be accessed as an attribute using given name.

Parameters

- **name** (*str*) – name of the parameter. The parameter can be accessed from this module using the given name
- **param** (*Parameter or None*) – parameter to be added to the module. If `None`, then operations that run on parameters, such as *cuda*, are ignored. If `None`, the parameter is **not** included in the module's *state_dict*.

requires_grad_(*requires_grad: bool = True*) → torch.nn.modules.module.T

Change if autograd should record operations on parameters in this module.

This method sets the parameters' `requires_grad` attributes in-place.

This method is helpful for freezing part of the module for finetuning or training parts of a model individually (e.g., GAN training).

See locally-disable-grad-doc for a comparison between *.requires_grad_()* and several similar mechanisms that may be confused with it.

Parameters **requires_grad** (*bool*) – whether autograd should record operations on parameters in this module. Default: `True`.

Returns self

Return type Module

set_extra_state(*state: Any*)
This function is called from *load_state_dict()* to handle any extra state found within the *state_dict*. Implement this function and a corresponding *get_extra_state()* for your module if you need to store extra state within its *state_dict*.

Parameters **state** (*dict*) – Extra state from the *state_dict*

share_memory() → torch.nn.modules.module.T
See `torch.Tensor.share_memory_()`

state_dict(**args*, *destination=None*, *prefix=''*, *keep_vars=False*)
Returns a dictionary containing references to the whole state of the module.

Both parameters and persistent buffers (e.g. running averages) are included. Keys are corresponding parameter and buffer names. Parameters and buffers set to `None` are not included.

Note: The returned object is a shallow copy. It contains references to the module's parameters and buffers.

Warning: Currently `state_dict()` also accepts positional arguments for `destination`, `prefix` and `keep_vars` in order. However, this is being deprecated and keyword arguments will be enforced in future releases.

Warning: Please avoid the use of argument `destination` as it is not designed for end-users.

Parameters

- **destination** (*dict, optional*) – If provided, the state of module will be updated into the dict and the same object is returned. Otherwise, an `OrderedDict` will be created and returned. Default: `None`.
- **prefix** (*str, optional*) – a prefix added to parameter and buffer names to compose the keys in state_dict. Default: `''`.
- **keep_vars** (*bool, optional*) – by default the `Tensor` s returned in the state dict are detached from autograd. If it's set to `True`, detaching will not be performed. Default: `False`.

Returns a dictionary containing a whole state of the module

Return type dict

Example:

```
>>> # xdoctest: +SKIP("undefined vars")
>>> module.state_dict().keys()
['bias', 'weight']
```

to(**args*, ***kwargs*)
Moves and/or casts the parameters and buffers.

This can be called as

to(*device=None*, *dtype=None*, *non_blocking=False*)

to(*dtype*, *non_blocking=False*)

to(*tensor*, *non_blocking=False*)

to(*memory_format=torch.channels_last*)

Its signature is similar to `torch.Tensor.to()`, but only accepts floating point or complex `dtype`s. In addition, this method will only cast the floating point or complex parameters and buffers to `dtype` (if given). The integral parameters and buffers will be moved `device`, if that is given, but with dtypes unchanged. When `non_blocking` is set, it tries to convert/move asynchronously with respect to the host if possible, e.g., moving CPU Tensors with pinned memory to CUDA devices.

See below for examples.

Note: This method modifies the module in-place.

Parameters

- **device** (`torch.device`) – the desired device of the parameters and buffers in this module
- **dtype** (`torch.dtype`) – the desired floating point or complex dtype of the parameters and buffers in this module
- **tensor** (*torch.Tensor*) – Tensor whose dtype and device are the desired dtype and device for all parameters and buffers in this module
- **memory_format** (`torch.memory_format`) – the desired memory format for 4D parameters and buffers in this module (keyword only argument)

Returns self

Return type Module

Examples:

```
>>> # xdoctest: +IGNORE_WANT("non-deterministic")
>>> linear = nn.Linear(2, 2)
>>> linear.weight
Parameter containing:
tensor([[ 0.1913, -0.3420],
        [-0.5113, -0.2325]])
>>> linear.to(torch.double)
Linear(in_features=2, out_features=2, bias=True)
>>> linear.weight
Parameter containing:
tensor([[ 0.1913, -0.3420],
        [-0.5113, -0.2325]], dtype=torch.float64)
>>> # xdoctest: +REQUIRES(env:TORCH_DOCTEST_CUDA1)
>>> gpu1 = torch.device("cuda:1")
>>> linear.to(gpu1, dtype=torch.half, non_blocking=True)
Linear(in_features=2, out_features=2, bias=True)
>>> linear.weight
Parameter containing:
tensor([[ 0.1914, -0.3420],
        [-0.5112, -0.2324]], dtype=torch.float16, device='cuda:1')
>>> cpu = torch.device("cpu")
>>> linear.to(cpu)
```

(continues on next page)

(continued from previous page)

```
Linear(in_features=2, out_features=2, bias=True)
>>> linear.weight
Parameter containing:
tensor([[ 0.1914, -0.3420],
        [-0.5112, -0.2324]], dtype=torch.float16)

>>> linear = nn.Linear(2, 2, bias=None).to(torch.cdouble)
>>> linear.weight
Parameter containing:
tensor([[ 0.3741+0.j,  0.2382+0.j],
        [ 0.5593+0.j, -0.4443+0.j]], dtype=torch.complex128)
>>> linear(torch.ones(3, 2, dtype=torch.cdouble))
tensor([[0.6122+0.j, 0.1150+0.j],
        [0.6122+0.j, 0.1150+0.j],
        [0.6122+0.j, 0.1150+0.j]], dtype=torch.complex128)
```

to_empty(*, *device: Union[str, torch.device]*) → torch.nn.modules.module.T
Moves the parameters and buffers to the specified device without copying storage.

Parameters **device** (`torch.device`) – The desired device of the parameters and buffers in this module.

Returns self

Return type Module

train(*mode: bool = True*) → torch.nn.modules.module.T
Sets the module in training mode.

This has any effect only on certain modules. See documentations of particular modules for details of their behaviors in training/evaluation mode, if they are affected, e.g. `Dropout`, `BatchNorm`, etc.

Parameters **mode** (*bool*) – whether to set training mode (`True`) or evaluation mode (`False`). Default: `True`.

Returns self

Return type Module

training: bool

type(*dst_type: Union[torch.dtype, str]*) → torch.nn.modules.module.T
Casts all parameters and buffers to `dst_type`.

Note: This method modifies the module in-place.

Parameters **dst_type** (*type or string*) – the desired type

Returns self

Return type Module

xpu(*device: Optional[Union[int, torch.device]] = None*) → torch.nn.modules.module.T
Moves all model parameters and buffers to the XPU.

This also makes associated parameters and buffers different objects. So it should be called before constructing optimizer if the module will live on XPU while being optimized.

Note: This method modifies the module in-place.

Parameters **device** (*int, optional*) – if specified, all parameters will be copied to that device

Returns self

Return type Module

zero_grad(*set_to_none: bool = False*) → None
Sets gradients of all model parameters to zero. See similar function under `torch.optim.Optimizer` for more context.

Parameters **set_to_none** (*bool*) – instead of setting to zero, set the grads to None. See `torch.optim.Optimizer.zero_grad()` for details.

synthcity.plugins.core.models.ts_vae module

class LatentODE(*n_units_embedding: int, n_layers_hidden: int = 1, n_units_hidden: int = 100, nonlin: str = 'relu', random_state: int = 0, dropout: float = 0.1, batch_norm: bool = True, residual: bool = False, device: Any = device(type='cpu')*)
Bases: `torch.nn.modules.module.Module`

T_destination
alias of TypeVar('T_destination', bound=`Dict[str, Any]`)

add_module(*name: str, module: Optional[torch.nn.modules.module.Module]*) → None
Adds a child module to the current module.

The module can be accessed as an attribute using the given name.

Parameters

- **name** (*str*) – name of the child module. The child module can be accessed from this module using the given name
- **module** (*Module*) – child module to be added to the module.

apply(*fn: Callable[[torch.nn.modules.module.Module], None]*) → torch.nn.modules.module.T
Applies `fn` recursively to every submodule (as returned by `.children()`) as well as self. Typical use includes initializing the parameters of a model (see also nn-init-doc).

Parameters **fn** (`Module` -> None) – function to be applied to each submodule

Returns self

Return type Module

Example:

```
>>> @torch.no_grad()
>>> def init_weights(m):
>>>     print(m)
>>>     if type(m) == nn.Linear:
>>>         m.weight.fill_(1.0)
>>>         print(m.weight)
>>> net = nn.Sequential(nn.Linear(2, 2), nn.Linear(2, 2))
>>> net.apply(init_weights)
```

(continues on next page)

(continued from previous page)

```
Linear(in_features=2, out_features=2, bias=True)
Parameter containing:
tensor([[1., 1.],
        [1., 1.]], requires_grad=True)
Linear(in_features=2, out_features=2, bias=True)
Parameter containing:
tensor([[1., 1.],
        [1., 1.]], requires_grad=True)
Sequential(
  (0): Linear(in_features=2, out_features=2, bias=True)
  (1): Linear(in_features=2, out_features=2, bias=True)
)
```

bfloat16() → torch.nn.modules.module.T
Casts all floating point parameters and buffers to `bfloat16` datatype.

Note: This method modifies the module in-place.

Returns self

Return type Module

buffers(*recurse: bool = True*) → Iterator[torch.Tensor]
Returns an iterator over module buffers.

Parameters **recurse** (*bool*) – if True, then yields buffers of this module and all submodules. Otherwise, yields only buffers that are direct members of this module.

Yields *torch.Tensor* – module buffer

Example:

```
>>> # xdoctest: +SKIP("undefined vars")
>>> for buf in model.buffers():
>>>     print(type(buf), buf.size())
<class 'torch.Tensor'> (20L,)
<class 'torch.Tensor'> (20L, 1L, 5L, 5L)
```

children() → Iterator[torch.nn.modules.module.Module]
Returns an iterator over immediate children modules.

Yields *Module* – a child module

cpu() → torch.nn.modules.module.T
Moves all model parameters and buffers to the CPU.

Note: This method modifies the module in-place.

Returns self

Return type Module

cuda(*device: Optional[Union[int, torch.device]] = None*) → torch.nn.modules.module.T
Moves all model parameters and buffers to the GPU.

This also makes associated parameters and buffers different objects. So it should be called before constructing optimizer if the module will live on GPU while being optimized.

Note: This method modifies the module in-place.

Parameters **device** (*int, optional*) – if specified, all parameters will be copied to that device

Returns self

Return type Module

double() → torch.nn.modules.module.T
Casts all floating point parameters and buffers to `double` datatype.

Note: This method modifies the module in-place.

Returns self

Return type Module

dump_patches: bool = False

eval() → torch.nn.modules.module.T
Sets the module in evaluation mode.

This has any effect only on certain modules. See documentations of particular modules for details of their behaviors in training/evaluation mode, if they are affected, e.g. `Dropout`, `BatchNorm`, etc.

This is equivalent with `self.train(False)`.

See locally-disable-grad-doc for a comparison between *.eval()* and several similar mechanisms that may be confused with it.

Returns self

Return type Module

extra_repr() → str
Set the extra representation of the module

To print customized extra information, you should re-implement this method in your own modules. Both single-line and multi-line strings are acceptable.

float() → torch.nn.modules.module.T
Casts all floating point parameters and buffers to `float` datatype.

Note: This method modifies the module in-place.

Returns self

Return type Module

forward(*observation_times: torch.Tensor*, *temporal: torch.Tensor*) → torch.Tensor
Defines the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

---

get_buffer(*target: str*) → torch.Tensor
Returns the buffer given by `target` if it exists, otherwise throws an error.

See the docstring for `get_submodule` for a more detailed explanation of this method's functionality as well as how to correctly specify `target`.

Parameters **target** – The fully-qualified string name of the buffer to look for. (See `get_submodule` for how to specify a fully-qualified string.)

Returns The buffer referenced by `target`

Return type torch.Tensor

Raises **AttributeError** – If the target string references an invalid path or resolves to something that is not a buffer

get_extra_state() → Any
Returns any extra state to include in the module's state_dict. Implement this and a corresponding *set_extra_state()* for your module if you need to store extra state. This function is called when building the module's *state_dict()*.

Note that extra state should be pickleable to ensure working serialization of the state_dict. We only provide provide backwards compatibility guarantees for serializing Tensors; other objects may break backwards compatibility if their serialized pickled form changes.

Returns Any extra state to store in the module's state_dict

Return type object

get_parameter(*target: str*) → torch.nn.parameter.Parameter
Returns the parameter given by `target` if it exists, otherwise throws an error.

See the docstring for `get_submodule` for a more detailed explanation of this method's functionality as well as how to correctly specify `target`.

Parameters **target** – The fully-qualified string name of the Parameter to look for. (See `get_submodule` for how to specify a fully-qualified string.)

Returns The Parameter referenced by `target`

Return type torch.nn.Parameter

Raises **AttributeError** – If the target string references an invalid path or resolves to something that is not an `nn.Parameter`

get_submodule(*target: str*) → torch.nn.modules.module.Module
Returns the submodule given by `target` if it exists, otherwise throws an error.

For example, let's say you have an `nn.Module` `A` that looks like this:

```
A(
    (net_b): Module(
```

(continues on next page)

(continued from previous page)

```
        (net_c): Module(
            (conv): Conv2d(16, 33, kernel_size=(3, 3), stride=(2, 2))
        )
        (linear): Linear(in_features=100, out_features=200, bias=True)
    )
)
```

(The diagram shows an `nn.Module` `A`. `A` has a nested submodule `net_b`, which itself has two submodules `net_c` and `linear`. `net_c` then has a submodule `conv`.)

To check whether or not we have the `linear` submodule, we would call `get_submodule("net_b.linear")`. To check whether we have the `conv` submodule, we would call `get_submodule("net_b.net_c.conv")`.

The runtime of `get_submodule` is bounded by the degree of module nesting in `target`. A query against `named_modules` achieves the same result, but it is O(N) in the number of transitive modules. So, for a simple check to see if some submodule exists, `get_submodule` should always be used.

Parameters `target` – The fully-qualified string name of the submodule to look for. (See above example for how to specify a fully-qualified string.)

Returns The submodule referenced by `target`

Return type torch.nn.Module

Raises `AttributeError` – If the target string references an invalid path or resolves to something that is not an `nn.Module`

`half`() → torch.nn.modules.module.T

Casts all floating point parameters and buffers to `half` datatype.

Note: This method modifies the module in-place.

Returns self

Return type Module

`ipu`(*device: Optional[Union[int, torch.device]] = None*) → torch.nn.modules.module.T

Moves all model parameters and buffers to the IPU.

This also makes associated parameters and buffers different objects. So it should be called before constructing optimizer if the module will live on IPU while being optimized.

Note: This method modifies the module in-place.

Parameters `device` (*int, optional*) – if specified, all parameters will be copied to that device

Returns self

Return type Module

load_state_dict(*state_dict: Mapping[str, Any]*, *strict: bool = True*)

Copies parameters and buffers from *state_dict* into this module and its descendants. If `strict` is `True`, then the keys of *state_dict* must exactly match the keys returned by this module's `state_dict()` function.

Parameters

- **state_dict** (*dict*) – a dict containing parameters and persistent buffers.
- **strict** (*bool, optional*) – whether to strictly enforce that the keys in *state_dict* match the keys returned by this module's `state_dict()` function. Default: `True`

Returns

- **missing_keys** is a list of str containing the missing keys
- **unexpected_keys** is a list of str containing the unexpected keys

Return type `NamedTuple` with `missing_keys` and `unexpected_keys` fields

Note: If a parameter or buffer is registered as `None` and its corresponding key exists in *state_dict*, *load_state_dict()* will raise a `RuntimeError`.

---

modules() → Iterator[torch.nn.modules.module.Module]

Returns an iterator over all modules in the network.

Yields *Module* – a module in the network

Note: Duplicate modules are returned only once. In the following example, `l` will be returned only once.

Example:

```
>>> l = nn.Linear(2, 2)
>>> net = nn.Sequential(l, l)
>>> for idx, m in enumerate(net.modules()):
...     print(idx, '->', m)

0 -> Sequential(
  (0): Linear(in_features=2, out_features=2, bias=True)
  (1): Linear(in_features=2, out_features=2, bias=True)
)
1 -> Linear(in_features=2, out_features=2, bias=True)
```

named_buffers(*prefix: str = ''*, *recurse: bool = True*) → Iterator[Tuple[str, torch.Tensor]]

Returns an iterator over module buffers, yielding both the name of the buffer as well as the buffer itself.

Parameters

- **prefix** (*str*) – prefix to prepend to all buffer names.
- **recurse** (*bool*) – if True, then yields buffers of this module and all submodules. Otherwise, yields only buffers that are direct members of this module.

Yields *(str, torch.Tensor)* – Tuple containing the name and buffer

Example:

```
>>> # xdoctest: +SKIP("undefined vars")
>>> for name, buf in self.named_buffers():
>>>    if name in ['running_var']:
>>>        print(buf.size())
```

named_children() → Iterator[Tuple[str, torch.nn.modules.module.Module]]

Returns an iterator over immediate children modules, yielding both the name of the module as well as the module itself.

Yields *(str, Module)* – Tuple containing a name and child module

Example:

```
>>> # xdoctest: +SKIP("undefined vars")
>>> for name, module in model.named_children():
>>>     if name in ['conv4', 'conv5']:
>>>         print(module)
```

named_modules(*memo: Optional[Set[torch.nn.modules.module.Module]] = None*, *prefix: str = ''*, *remove_duplicate: bool = True*)

Returns an iterator over all modules in the network, yielding both the name of the module as well as the module itself.

Parameters

- **memo** – a memo to store the set of modules already added to the result
- **prefix** – a prefix that will be added to the name of the module
- **remove_duplicate** – whether to remove the duplicated module instances in the result or not

Yields *(str, Module)* – Tuple of name and module

Note: Duplicate modules are returned only once. In the following example, `l` will be returned only once.

Example:

```
>>> l = nn.Linear(2, 2)
>>> net = nn.Sequential(l, l)
>>> for idx, m in enumerate(net.named_modules()):
...     print(idx, '->', m)

0 -> ('', Sequential(
  (0): Linear(in_features=2, out_features=2, bias=True)
  (1): Linear(in_features=2, out_features=2, bias=True)
))
1 -> ('0', Linear(in_features=2, out_features=2, bias=True))
```

named_parameters(*prefix: str = ''*, *recurse: bool = True*) → Iterator[Tuple[str, torch.nn.parameter.Parameter]]

Returns an iterator over module parameters, yielding both the name of the parameter as well as the parameter itself.

Parameters

- **prefix** (*str*) – prefix to prepend to all parameter names.

- **recurse** (*bool*) – if True, then yields parameters of this module and all submodules. Otherwise, yields only parameters that are direct members of this module.

Yields *(str, Parameter)* – Tuple containing the name and parameter

Example:

```
>>> # xdoctest: +SKIP("undefined vars")
>>> for name, param in self.named_parameters():
>>>    if name in ['bias']:
>>>        print(param.size())
```

parameters(*recurse: bool = True*) → Iterator[torch.nn.parameter.Parameter]

Returns an iterator over module parameters.

This is typically passed to an optimizer.

Parameters **recurse** (*bool*) – if True, then yields parameters of this module and all submodules. Otherwise, yields only parameters that are direct members of this module.

Yields *Parameter* – module parameter

Example:

```
>>> # xdoctest: +SKIP("undefined vars")
>>> for param in model.parameters():
>>>     print(type(param), param.size())
<class 'torch.Tensor'> (20L,)
<class 'torch.Tensor'> (20L, 1L, 5L, 5L)
```

register_backward_hook(*hook: Callable[[torch.nn.modules.module.Module, Union[Tuple[torch.Tensor, ...], torch.Tensor], Union[Tuple[torch.Tensor, ...], torch.Tensor]], Union[None, torch.Tensor]]*) → torch.utils.hooks.RemovableHandle

Registers a backward hook on the module.

This function is deprecated in favor of `register_full_backward_hook()` and the behavior of this function will change in future versions.

Returns a handle that can be used to remove the added hook by calling `handle.remove()`

Return type `torch.utils.hooks.RemovableHandle`

register_buffer(*name: str*, *tensor: Optional[torch.Tensor]*, *persistent: bool = True*) → None

Adds a buffer to the module.

This is typically used to register a buffer that should not to be considered a model parameter. For example, BatchNorm's `running_mean` is not a parameter, but is part of the module's state. Buffers, by default, are persistent and will be saved alongside parameters. This behavior can be changed by setting `persistent` to `False`. The only difference between a persistent buffer and a non-persistent buffer is that the latter will not be a part of this module's *state_dict*.

Buffers can be accessed as attributes using given names.

Parameters

- **name** (*str*) – name of the buffer. The buffer can be accessed from this module using the given name
- **tensor** (*Tensor or None*) – buffer to be registered. If `None`, then operations that run on buffers, such as *cuda*, are ignored. If `None`, the buffer is **not** included in the module's *state_dict*.

- **persistent** (*bool*) – whether the buffer is part of this module's *state_dict*.

Example:

```
>>> # xdoctest: +SKIP("undefined vars")
>>> self.register_buffer('running_mean', torch.zeros(num_features))
```

register_forward_hook(*hook: Callable[[...], None]*) → torch.utils.hooks.RemovableHandle

Registers a forward hook on the module.

The hook will be called every time after *forward()* has computed an output. It should have the following signature:

```
hook(module, input, output) -> None or modified output
```

The input contains only the positional arguments given to the module. Keyword arguments won't be passed to the hooks and only to the `forward`. The hook can modify the output. It can modify the input inplace but it will not have effect on forward since this is called after *forward()* is called.

Returns a handle that can be used to remove the added hook by calling `handle.remove()`

Return type `torch.utils.hooks.RemovableHandle`

register_forward_pre_hook(*hook: Callable[[...], None]*) → torch.utils.hooks.RemovableHandle

Registers a forward pre-hook on the module.

The hook will be called every time before *forward()* is invoked. It should have the following signature:

```
hook(module, input) -> None or modified input
```

The input contains only the positional arguments given to the module. Keyword arguments won't be passed to the hooks and only to the `forward`. The hook can modify the input. User can either return a tuple or a single modified value in the hook. We will wrap the value into a tuple if a single value is returned(unless that value is already a tuple).

Returns a handle that can be used to remove the added hook by calling `handle.remove()`

Return type `torch.utils.hooks.RemovableHandle`

register_full_backward_hook(*hook: Callable[[torch.nn.modules.module.Module, Union[Tuple[torch.Tensor, ...], torch.Tensor], Union[Tuple[torch.Tensor, ...], torch.Tensor]], Union[None, torch.Tensor]]*) → torch.utils.hooks.RemovableHandle

Registers a backward hook on the module.

The hook will be called every time the gradients with respect to a module are computed, i.e. the hook will execute if and only if the gradients with respect to module outputs are computed. The hook should have the following signature:

```
hook(module, grad_input, grad_output) -> tuple(Tensor) or None
```

The `grad_input` and `grad_output` are tuples that contain the gradients with respect to the inputs and outputs respectively. The hook should not modify its arguments, but it can optionally return a new gradient with respect to the input that will be used in place of `grad_input` in subsequent computations. `grad_input` will only correspond to the inputs given as positional arguments and all kwarg arguments are ignored. Entries in `grad_input` and `grad_output` will be `None` for all non-Tensor arguments.

For technical reasons, when this hook is applied to a Module, its forward function will receive a view of each Tensor passed to the Module. Similarly the caller will receive a view of each Tensor returned by the Module's forward function.

> **Warning:** Modifying inputs or outputs inplace is not allowed when using backward hooks and will raise an error.

Returns a handle that can be used to remove the added hook by calling `handle.remove()`

Return type `torch.utils.hooks.RemovableHandle`

register_load_state_dict_post_hook(*hook*)

Registers a post hook to be run after module's `load_state_dict` is called.

It should have the following signature:: hook(module, incompatible_keys) -> None

The `module` argument is the current module that this hook is registered on, and the `incompatible_keys` argument is a `NamedTuple` consisting of attributes `missing_keys` and `unexpected_keys`. `missing_keys` is a `list` of `str` containing the missing keys and `unexpected_keys` is a `list` of `str` containing the unexpected keys.

The given incompatible_keys can be modified inplace if needed.

Note that the checks performed when calling *load_state_dict()* with `strict=True` are affected by modifications the hook makes to `missing_keys` or `unexpected_keys`, as expected. Additions to either set of keys will result in an error being thrown when `strict=True`, and clearning out both missing and unexpected keys will avoid an error.

Returns a handle that can be used to remove the added hook by calling `handle.remove()`

Return type `torch.utils.hooks.RemovableHandle`

register_module(*name: str*, *module: Optional[torch.nn.modules.module.Module]*) → None

Alias for *add_module()*.

register_parameter(*name: str*, *param: Optional[torch.nn.parameter.Parameter]*) → None

Adds a parameter to the module.

The parameter can be accessed as an attribute using given name.

Parameters

- **name** (*str*) – name of the parameter. The parameter can be accessed from this module using the given name
- **param** (*Parameter or None*) – parameter to be added to the module. If `None`, then operations that run on parameters, such as *cuda*, are ignored. If `None`, the parameter is **not** included in the module's *state_dict*.

requires_grad_(*requires_grad: bool = True*) → torch.nn.modules.module.T

Change if autograd should record operations on parameters in this module.

This method sets the parameters' `requires_grad` attributes in-place.

This method is helpful for freezing part of the module for finetuning or training parts of a model individually (e.g., GAN training).

See locally-disable-grad-doc for a comparison between *.requires_grad_()* and several similar mechanisms that may be confused with it.

Parameters **requires_grad** (*bool*) – whether autograd should record operations on parameters in this module. Default: `True`.

Returns self

Return type Module

set_extra_state(*state: Any*)
This function is called from *load_state_dict()* to handle any extra state found within the *state_dict*. Implement this function and a corresponding *get_extra_state()* for your module if you need to store extra state within its *state_dict*.

Parameters **state** (*dict*) – Extra state from the *state_dict*

share_memory() → torch.nn.modules.module.T
See `torch.Tensor.share_memory_()`

state_dict(**args*, *destination=None*, *prefix=''*, *keep_vars=False*)
Returns a dictionary containing references to the whole state of the module.

Both parameters and persistent buffers (e.g. running averages) are included. Keys are corresponding parameter and buffer names. Parameters and buffers set to `None` are not included.

Note: The returned object is a shallow copy. It contains references to the module's parameters and buffers.

Warning: Currently `state_dict()` also accepts positional arguments for `destination`, `prefix` and `keep_vars` in order. However, this is being deprecated and keyword arguments will be enforced in future releases.

Warning: Please avoid the use of argument `destination` as it is not designed for end-users.

Parameters

- **destination** (*dict, optional*) – If provided, the state of module will be updated into the dict and the same object is returned. Otherwise, an `OrderedDict` will be created and returned. Default: `None`.
- **prefix** (*str, optional*) – a prefix added to parameter and buffer names to compose the keys in state_dict. Default: `''`.
- **keep_vars** (*bool, optional*) – by default the `Tensor` s returned in the state dict are detached from autograd. If it's set to `True`, detaching will not be performed. Default: `False`.

Returns a dictionary containing a whole state of the module

Return type dict

Example:

```
>>> # xdoctest: +SKIP("undefined vars")
>>> module.state_dict().keys()
['bias', 'weight']
```

to(**args*, ***kwargs*)
Moves and/or casts the parameters and buffers.

This can be called as

to(*device=None*, *dtype=None*, *non_blocking=False*)

to(*dtype*, *non_blocking=False*)

to(*tensor*, *non_blocking=False*)

to(*memory_format=torch.channels_last*)

Its signature is similar to `torch.Tensor.to()`, but only accepts floating point or complex `dtype`s. In addition, this method will only cast the floating point or complex parameters and buffers to `dtype` (if given). The integral parameters and buffers will be moved `device`, if that is given, but with dtypes unchanged. When `non_blocking` is set, it tries to convert/move asynchronously with respect to the host if possible, e.g., moving CPU Tensors with pinned memory to CUDA devices.

See below for examples.

Note: This method modifies the module in-place.

Parameters

- **device** (`torch.device`) – the desired device of the parameters and buffers in this module
- **dtype** (`torch.dtype`) – the desired floating point or complex dtype of the parameters and buffers in this module
- **tensor** (*torch.Tensor*) – Tensor whose dtype and device are the desired dtype and device for all parameters and buffers in this module
- **memory_format** (`torch.memory_format`) – the desired memory format for 4D parameters and buffers in this module (keyword only argument)

Returns self

Return type Module

Examples:

```
>>> # xdoctest: +IGNORE_WANT("non-deterministic")
>>> linear = nn.Linear(2, 2)
>>> linear.weight
Parameter containing:
tensor([[ 0.1913, -0.3420],
        [-0.5113, -0.2325]])
>>> linear.to(torch.double)
Linear(in_features=2, out_features=2, bias=True)
>>> linear.weight
Parameter containing:
tensor([[ 0.1913, -0.3420],
        [-0.5113, -0.2325]], dtype=torch.float64)
>>> # xdoctest: +REQUIRES(env:TORCH_DOCTEST_CUDA1)
>>> gpu1 = torch.device("cuda:1")
>>> linear.to(gpu1, dtype=torch.half, non_blocking=True)
Linear(in_features=2, out_features=2, bias=True)
>>> linear.weight
Parameter containing:
tensor([[ 0.1914, -0.3420],
        [-0.5112, -0.2324]], dtype=torch.float16, device='cuda:1')
>>> cpu = torch.device("cpu")
>>> linear.to(cpu)
```

(continues on next page)

(continued from previous page)

```
Linear(in_features=2, out_features=2, bias=True)
>>> linear.weight
Parameter containing:
tensor([[ 0.1914, -0.3420],
        [-0.5112, -0.2324]], dtype=torch.float16)

>>> linear = nn.Linear(2, 2, bias=None).to(torch.cdouble)
>>> linear.weight
Parameter containing:
tensor([[ 0.3741+0.j,  0.2382+0.j],
        [ 0.5593+0.j, -0.4443+0.j]], dtype=torch.complex128)
>>> linear(torch.ones(3, 2, dtype=torch.cdouble))
tensor([[0.6122+0.j, 0.1150+0.j],
        [0.6122+0.j, 0.1150+0.j],
        [0.6122+0.j, 0.1150+0.j]], dtype=torch.complex128)
```

to_empty(*, *device: Union[str, torch.device]*) → torch.nn.modules.module.T
Moves the parameters and buffers to the specified device without copying storage.

Parameters **device** (`torch.device`) – The desired device of the parameters and buffers in this module.

Returns self

Return type Module

train(*mode: bool = True*) → torch.nn.modules.module.T
Sets the module in training mode.

This has any effect only on certain modules. See documentations of particular modules for details of their behaviors in training/evaluation mode, if they are affected, e.g. `Dropout`, `BatchNorm`, etc.

Parameters **mode** (*bool*) – whether to set training mode (`True`) or evaluation mode (`False`). Default: `True`.

Returns self

Return type Module

training: bool

type(*dst_type: Union[torch.dtype, str]*) → torch.nn.modules.module.T
Casts all parameters and buffers to `dst_type`.

Note: This method modifies the module in-place.

Parameters **dst_type** (*type or string*) – the desired type

Returns self

Return type Module

xpu(*device: Optional[Union[int, torch.device]] = None*) → torch.nn.modules.module.T
Moves all model parameters and buffers to the XPU.

This also makes associated parameters and buffers different objects. So it should be called before constructing optimizer if the module will live on XPU while being optimized.

Note: This method modifies the module in-place.

Parameters **device** (*int, optional*) – if specified, all parameters will be copied to that device

Returns self

Return type Module

zero_grad(*set_to_none: bool = False*) → None
Sets gradients of all model parameters to zero. See similar function under `torch.optim.Optimizer` for more context.

Parameters **set_to_none** (*bool*) – instead of setting to zero, set the grads to None. See `torch.optim.Optimizer.zero_grad()` for details.

class TimeSeriesDecoder(*n_static_units_embedding: int, n_static_units_out: int, n_temporal_units_embedding: int, n_temporal_units_out: int, n_temporal_window: int, n_layers_hidden: int = 1, n_units_hidden: int = 100, nonlin: str = 'relu', static_nonlin_out: Optional[List[Tuple[str, int]]] = None, temporal_nonlin_out: Optional[List[Tuple[str, int]]] = None, random_state: int = 0, dropout: float = 0.1, mode: str = 'LSTM', batch_norm: bool = False, residual: bool = False, device: Any = device(type='cpu')*)

Bases: `torch.nn.modules.module.Module`

T_destination
alias of TypeVar('T_destination', bound=`Dict[str, Any]`)

add_module(*name: str, module: Optional[torch.nn.modules.module.Module]*) → None
Adds a child module to the current module.

The module can be accessed as an attribute using the given name.

Parameters

- **name** (*str*) – name of the child module. The child module can be accessed from this module using the given name
- **module** (*Module*) – child module to be added to the module.

apply(*fn: Callable[[torch.nn.modules.module.Module], None]*) → torch.nn.modules.module.T
Applies `fn` recursively to every submodule (as returned by `.children()`) as well as self. Typical use includes initializing the parameters of a model (see also nn-init-doc).

Parameters **fn** (`Module` -> None) – function to be applied to each submodule

Returns self

Return type Module

Example:

```
>>> @torch.no_grad()
>>> def init_weights(m):
>>>     print(m)
>>>     if type(m) == nn.Linear:
>>>         m.weight.fill_(1.0)
>>>         print(m.weight)
>>> net = nn.Sequential(nn.Linear(2, 2), nn.Linear(2, 2))
```

(continues on next page)

(continued from previous page)

```
>>> net.apply(init_weights)
Linear(in_features=2, out_features=2, bias=True)
Parameter containing:
tensor([[1., 1.],
        [1., 1.]], requires_grad=True)
Linear(in_features=2, out_features=2, bias=True)
Parameter containing:
tensor([[1., 1.],
        [1., 1.]], requires_grad=True)
Sequential(
  (0): Linear(in_features=2, out_features=2, bias=True)
  (1): Linear(in_features=2, out_features=2, bias=True)
)
```

bfloat16() → torch.nn.modules.module.T
Casts all floating point parameters and buffers to `bfloat16` datatype.

Note: This method modifies the module in-place.

Returns self

Return type Module

buffers(*recurse: bool = True*) → Iterator[torch.Tensor]
Returns an iterator over module buffers.

Parameters **recurse** (*bool*) – if True, then yields buffers of this module and all submodules. Otherwise, yields only buffers that are direct members of this module.

Yields *torch.Tensor* – module buffer

Example:

```
>>> # xdoctest: +SKIP("undefined vars")
>>> for buf in model.buffers():
>>>     print(type(buf), buf.size())
<class 'torch.Tensor'> (20L,)
<class 'torch.Tensor'> (20L, 1L, 5L, 5L)
```

children() → Iterator[torch.nn.modules.module.Module]
Returns an iterator over immediate children modules.

Yields *Module* – a child module

cpu() → torch.nn.modules.module.T
Moves all model parameters and buffers to the CPU.

Note: This method modifies the module in-place.

Returns self

Return type Module

cuda(*device: Optional[Union[int, torch.device]] = None*) → torch.nn.modules.module.T
Moves all model parameters and buffers to the GPU.

This also makes associated parameters and buffers different objects. So it should be called before constructing optimizer if the module will live on GPU while being optimized.

Note: This method modifies the module in-place.

Parameters **device** (*int, optional*) – if specified, all parameters will be copied to that device

Returns self

Return type Module

double() → torch.nn.modules.module.T
Casts all floating point parameters and buffers to `double` datatype.

Note: This method modifies the module in-place.

Returns self

Return type Module

dump_patches: bool = False

eval() → torch.nn.modules.module.T
Sets the module in evaluation mode.

This has any effect only on certain modules. See documentations of particular modules for details of their behaviors in training/evaluation mode, if they are affected, e.g. `Dropout`, `BatchNorm`, etc.

This is equivalent with `self.train(False)`.

See locally-disable-grad-doc for a comparison between *.eval()* and several similar mechanisms that may be confused with it.

Returns self

Return type Module

extra_repr() → str
Set the extra representation of the module

To print customized extra information, you should re-implement this method in your own modules. Both single-line and multi-line strings are acceptable.

float() → torch.nn.modules.module.T
Casts all floating point parameters and buffers to `float` datatype.

Note: This method modifies the module in-place.

Returns self

Return type Module

forward(*static_embs: torch.Tensor*, *temporal_embs: torch.Tensor*, *horizon_embs: torch.Tensor*) → torch.Tensor

Defines the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

---

get_buffer(*target: str*) → torch.Tensor

Returns the buffer given by `target` if it exists, otherwise throws an error.

See the docstring for `get_submodule` for a more detailed explanation of this method's functionality as well as how to correctly specify `target`.

Parameters **target** – The fully-qualified string name of the buffer to look for. (See `get_submodule` for how to specify a fully-qualified string.)

Returns The buffer referenced by `target`

Return type torch.Tensor

Raises **AttributeError** – If the target string references an invalid path or resolves to something that is not a buffer

get_extra_state() → Any

Returns any extra state to include in the module's state_dict. Implement this and a corresponding *set_extra_state()* for your module if you need to store extra state. This function is called when building the module's *state_dict()*.

Note that extra state should be pickleable to ensure working serialization of the state_dict. We only provide provide backwards compatibility guarantees for serializing Tensors; other objects may break backwards compatibility if their serialized pickled form changes.

Returns Any extra state to store in the module's state_dict

Return type object

get_parameter(*target: str*) → torch.nn.parameter.Parameter

Returns the parameter given by `target` if it exists, otherwise throws an error.

See the docstring for `get_submodule` for a more detailed explanation of this method's functionality as well as how to correctly specify `target`.

Parameters **target** – The fully-qualified string name of the Parameter to look for. (See `get_submodule` for how to specify a fully-qualified string.)

Returns The Parameter referenced by `target`

Return type torch.nn.Parameter

Raises **AttributeError** – If the target string references an invalid path or resolves to something that is not an `nn.Parameter`

get_submodule(*target: str*) → torch.nn.modules.module.Module

Returns the submodule given by `target` if it exists, otherwise throws an error.

For example, let's say you have an `nn.Module A` that looks like this:

```
A(
    (net_b): Module(
        (net_c): Module(
            (conv): Conv2d(16, 33, kernel_size=(3, 3), stride=(2, 2))
        )
        (linear): Linear(in_features=100, out_features=200, bias=True)
    )
)
```

(The diagram shows an `nn.Module` `A`. `A` has a nested submodule `net_b`, which itself has two submodules `net_c` and `linear`. `net_c` then has a submodule `conv`.)

To check whether or not we have the `linear` submodule, we would call `get_submodule("net_b.linear")`. To check whether we have the `conv` submodule, we would call `get_submodule("net_b.net_c.conv")`.

The runtime of `get_submodule` is bounded by the degree of module nesting in `target`. A query against `named_modules` achieves the same result, but it is O(N) in the number of transitive modules. So, for a simple check to see if some submodule exists, `get_submodule` should always be used.

Parameters `target` – The fully-qualified string name of the submodule to look for. (See above example for how to specify a fully-qualified string.)

Returns The submodule referenced by `target`

Return type torch.nn.Module

Raises `AttributeError` – If the target string references an invalid path or resolves to something that is not an `nn.Module`

`half`() → torch.nn.modules.module.T

Casts all floating point parameters and buffers to `half` datatype.

Note: This method modifies the module in-place.

Returns self

Return type Module

`ipu`(*device: Optional[Union[int, torch.device]] = None*) → torch.nn.modules.module.T

Moves all model parameters and buffers to the IPU.

This also makes associated parameters and buffers different objects. So it should be called before constructing optimizer if the module will live on IPU while being optimized.

Note: This method modifies the module in-place.

Parameters `device` (*int, optional*) – if specified, all parameters will be copied to that device

Returns self

Return type Module

load_state_dict(*state_dict: Mapping[str, Any]*, *strict: bool = True*)

Copies parameters and buffers from *state_dict* into this module and its descendants. If `strict` is `True`, then the keys of *state_dict* must exactly match the keys returned by this module's `state_dict()` function.

Parameters

- **state_dict** (*dict*) – a dict containing parameters and persistent buffers.
- **strict** (*bool, optional*) – whether to strictly enforce that the keys in *state_dict* match the keys returned by this module's `state_dict()` function. Default: `True`

Returns

- **missing_keys** is a list of str containing the missing keys
- **unexpected_keys** is a list of str containing the unexpected keys

Return type `NamedTuple` with `missing_keys` and `unexpected_keys` fields

Note: If a parameter or buffer is registered as `None` and its corresponding key exists in *state_dict*, *load_state_dict()* will raise a `RuntimeError`.

---

modules() → Iterator[torch.nn.modules.module.Module]

Returns an iterator over all modules in the network.

Yields *Module* – a module in the network

Note: Duplicate modules are returned only once. In the following example, `l` will be returned only once.

Example:

```
>>> l = nn.Linear(2, 2)
>>> net = nn.Sequential(l, l)
>>> for idx, m in enumerate(net.modules()):
...     print(idx, '->', m)

0 -> Sequential(
  (0): Linear(in_features=2, out_features=2, bias=True)
  (1): Linear(in_features=2, out_features=2, bias=True)
)
1 -> Linear(in_features=2, out_features=2, bias=True)
```

named_buffers(*prefix: str = ''*, *recurse: bool = True*) → Iterator[Tuple[str, torch.Tensor]]

Returns an iterator over module buffers, yielding both the name of the buffer as well as the buffer itself.

Parameters

- **prefix** (*str*) – prefix to prepend to all buffer names.
- **recurse** (*bool*) – if True, then yields buffers of this module and all submodules. Otherwise, yields only buffers that are direct members of this module.

Yields *(str, torch.Tensor)* – Tuple containing the name and buffer

Example:

```
>>> # xdoctest: +SKIP("undefined vars")
>>> for name, buf in self.named_buffers():
>>>    if name in ['running_var']:
>>>        print(buf.size())
```

named_children() → Iterator[Tuple[str, torch.nn.modules.module.Module]]

Returns an iterator over immediate children modules, yielding both the name of the module as well as the module itself.

Yields *(str, Module)* – Tuple containing a name and child module

Example:

```
>>> # xdoctest: +SKIP("undefined vars")
>>> for name, module in model.named_children():
>>>     if name in ['conv4', 'conv5']:
>>>         print(module)
```

named_modules(*memo: Optional[Set[torch.nn.modules.module.Module]] = None*, *prefix: str = ''*, *remove_duplicate: bool = True*)

Returns an iterator over all modules in the network, yielding both the name of the module as well as the module itself.

Parameters

- **memo** – a memo to store the set of modules already added to the result
- **prefix** – a prefix that will be added to the name of the module
- **remove_duplicate** – whether to remove the duplicated module instances in the result or not

Yields *(str, Module)* – Tuple of name and module

Note: Duplicate modules are returned only once. In the following example, `l` will be returned only once.

Example:

```
>>> l = nn.Linear(2, 2)
>>> net = nn.Sequential(l, l)
>>> for idx, m in enumerate(net.named_modules()):
...     print(idx, '->', m)

0 -> ('', Sequential(
  (0): Linear(in_features=2, out_features=2, bias=True)
  (1): Linear(in_features=2, out_features=2, bias=True)
))
1 -> ('0', Linear(in_features=2, out_features=2, bias=True))
```

named_parameters(*prefix: str = ''*, *recurse: bool = True*) → Iterator[Tuple[str, torch.nn.parameter.Parameter]]

Returns an iterator over module parameters, yielding both the name of the parameter as well as the parameter itself.

Parameters

- **prefix** (*str*) – prefix to prepend to all parameter names.

- **recurse** (*bool*) – if True, then yields parameters of this module and all submodules. Otherwise, yields only parameters that are direct members of this module.

Yields *(str, Parameter)* – Tuple containing the name and parameter

Example:

```
>>> # xdoctest: +SKIP("undefined vars")
>>> for name, param in self.named_parameters():
>>>    if name in ['bias']:
>>>        print(param.size())
```

parameters(*recurse: bool = True*) → Iterator[torch.nn.parameter.Parameter]

Returns an iterator over module parameters.

This is typically passed to an optimizer.

Parameters **recurse** (*bool*) – if True, then yields parameters of this module and all submodules. Otherwise, yields only parameters that are direct members of this module.

Yields *Parameter* – module parameter

Example:

```
>>> # xdoctest: +SKIP("undefined vars")
>>> for param in model.parameters():
>>>     print(type(param), param.size())
<class 'torch.Tensor'> (20L,)
<class 'torch.Tensor'> (20L, 1L, 5L, 5L)
```

register_backward_hook(*hook: Callable[[torch.nn.modules.module.Module, Union[Tuple[torch.Tensor, ...], torch.Tensor], Union[Tuple[torch.Tensor, ...], torch.Tensor]], Union[None, torch.Tensor]]*) → torch.utils.hooks.RemovableHandle

Registers a backward hook on the module.

This function is deprecated in favor of `register_full_backward_hook()` and the behavior of this function will change in future versions.

Returns a handle that can be used to remove the added hook by calling `handle.remove()`

Return type `torch.utils.hooks.RemovableHandle`

register_buffer(*name: str*, *tensor: Optional[torch.Tensor]*, *persistent: bool = True*) → None

Adds a buffer to the module.

This is typically used to register a buffer that should not to be considered a model parameter. For example, BatchNorm's `running_mean` is not a parameter, but is part of the module's state. Buffers, by default, are persistent and will be saved alongside parameters. This behavior can be changed by setting `persistent` to `False`. The only difference between a persistent buffer and a non-persistent buffer is that the latter will not be a part of this module's *state_dict*.

Buffers can be accessed as attributes using given names.

Parameters

- **name** (*str*) – name of the buffer. The buffer can be accessed from this module using the given name
- **tensor** (*Tensor or None*) – buffer to be registered. If `None`, then operations that run on buffers, such as *cuda*, are ignored. If `None`, the buffer is **not** included in the module's *state_dict*.

- **persistent** (*bool*) – whether the buffer is part of this module's *state_dict*.

Example:

```
>>> # xdoctest: +SKIP("undefined vars")
>>> self.register_buffer('running_mean', torch.zeros(num_features))
```

register_forward_hook(*hook: Callable[[...], None]*) → torch.utils.hooks.RemovableHandle

Registers a forward hook on the module.

The hook will be called every time after *forward()* has computed an output. It should have the following signature:

```
hook(module, input, output) -> None or modified output
```

The input contains only the positional arguments given to the module. Keyword arguments won't be passed to the hooks and only to the `forward`. The hook can modify the output. It can modify the input inplace but it will not have effect on forward since this is called after *forward()* is called.

Returns a handle that can be used to remove the added hook by calling `handle.remove()`

Return type `torch.utils.hooks.RemovableHandle`

register_forward_pre_hook(*hook: Callable[[...], None]*) → torch.utils.hooks.RemovableHandle

Registers a forward pre-hook on the module.

The hook will be called every time before *forward()* is invoked. It should have the following signature:

```
hook(module, input) -> None or modified input
```

The input contains only the positional arguments given to the module. Keyword arguments won't be passed to the hooks and only to the `forward`. The hook can modify the input. User can either return a tuple or a single modified value in the hook. We will wrap the value into a tuple if a single value is returned(unless that value is already a tuple).

Returns a handle that can be used to remove the added hook by calling `handle.remove()`

Return type `torch.utils.hooks.RemovableHandle`

register_full_backward_hook(*hook: Callable[[torch.nn.modules.module.Module, Union[Tuple[torch.Tensor, ...], torch.Tensor], Union[Tuple[torch.Tensor, ...], torch.Tensor]], Union[None, torch.Tensor]]*) → torch.utils.hooks.RemovableHandle

Registers a backward hook on the module.

The hook will be called every time the gradients with respect to a module are computed, i.e. the hook will execute if and only if the gradients with respect to module outputs are computed. The hook should have the following signature:

```
hook(module, grad_input, grad_output) -> tuple(Tensor) or None
```

The `grad_input` and `grad_output` are tuples that contain the gradients with respect to the inputs and outputs respectively. The hook should not modify its arguments, but it can optionally return a new gradient with respect to the input that will be used in place of `grad_input` in subsequent computations. `grad_input` will only correspond to the inputs given as positional arguments and all kwarg arguments are ignored. Entries in `grad_input` and `grad_output` will be `None` for all non-Tensor arguments.

For technical reasons, when this hook is applied to a Module, its forward function will receive a view of each Tensor passed to the Module. Similarly the caller will receive a view of each Tensor returned by the Module's forward function.

> **Warning:** Modifying inputs or outputs inplace is not allowed when using backward hooks and will raise an error.

Returns a handle that can be used to remove the added hook by calling `handle.remove()`

Return type `torch.utils.hooks.RemovableHandle`

register_load_state_dict_post_hook(*hook*)

Registers a post hook to be run after module's `load_state_dict` is called.

It should have the following signature:: hook(module, incompatible_keys) -> None

The `module` argument is the current module that this hook is registered on, and the `incompatible_keys` argument is a `NamedTuple` consisting of attributes `missing_keys` and `unexpected_keys`. `missing_keys` is a `list` of `str` containing the missing keys and `unexpected_keys` is a `list` of `str` containing the unexpected keys.

The given incompatible_keys can be modified inplace if needed.

Note that the checks performed when calling *load_state_dict()* with `strict=True` are affected by modifications the hook makes to `missing_keys` or `unexpected_keys`, as expected. Additions to either set of keys will result in an error being thrown when `strict=True`, and clearning out both missing and unexpected keys will avoid an error.

Returns a handle that can be used to remove the added hook by calling `handle.remove()`

Return type `torch.utils.hooks.RemovableHandle`

register_module(*name: str*, *module: Optional[torch.nn.modules.module.Module]*) → None

Alias for *add_module()*.

register_parameter(*name: str*, *param: Optional[torch.nn.parameter.Parameter]*) → None

Adds a parameter to the module.

The parameter can be accessed as an attribute using given name.

Parameters

- **name** (*str*) – name of the parameter. The parameter can be accessed from this module using the given name
- **param** (*Parameter or None*) – parameter to be added to the module. If `None`, then operations that run on parameters, such as *cuda*, are ignored. If `None`, the parameter is **not** included in the module's *state_dict*.

requires_grad_(*requires_grad: bool = True*) → torch.nn.modules.module.T

Change if autograd should record operations on parameters in this module.

This method sets the parameters' `requires_grad` attributes in-place.

This method is helpful for freezing part of the module for finetuning or training parts of a model individually (e.g., GAN training).

See locally-disable-grad-doc for a comparison between *.requires_grad_()* and several similar mechanisms that may be confused with it.

Parameters **requires_grad** (*bool*) – whether autograd should record operations on parameters in this module. Default: `True`.

Returns self

Return type Module

set_extra_state(*state: Any*)
This function is called from *load_state_dict()* to handle any extra state found within the *state_dict*. Implement this function and a corresponding *get_extra_state()* for your module if you need to store extra state within its *state_dict*.

Parameters **state** (*dict*) – Extra state from the *state_dict*

share_memory() → torch.nn.modules.module.T
See `torch.Tensor.share_memory_()`

state_dict(**args*, *destination=None*, *prefix=''*, *keep_vars=False*)
Returns a dictionary containing references to the whole state of the module.

Both parameters and persistent buffers (e.g. running averages) are included. Keys are corresponding parameter and buffer names. Parameters and buffers set to `None` are not included.

Note: The returned object is a shallow copy. It contains references to the module's parameters and buffers.

Warning: Currently `state_dict()` also accepts positional arguments for `destination`, `prefix` and `keep_vars` in order. However, this is being deprecated and keyword arguments will be enforced in future releases.

Warning: Please avoid the use of argument `destination` as it is not designed for end-users.

Parameters

- **destination** (*dict, optional*) – If provided, the state of module will be updated into the dict and the same object is returned. Otherwise, an `OrderedDict` will be created and returned. Default: `None`.
- **prefix** (*str, optional*) – a prefix added to parameter and buffer names to compose the keys in state_dict. Default: `''`.
- **keep_vars** (*bool, optional*) – by default the `Tensor` s returned in the state dict are detached from autograd. If it's set to `True`, detaching will not be performed. Default: `False`.

Returns a dictionary containing a whole state of the module

Return type dict

Example:

```
>>> # xdoctest: +SKIP("undefined vars")
>>> module.state_dict().keys()
['bias', 'weight']
```

to(**args*, ***kwargs*)
Moves and/or casts the parameters and buffers.

This can be called as

to(*device=None*, *dtype=None*, *non_blocking=False*)

to(*dtype*, *non_blocking=False*)

to(*tensor*, *non_blocking=False*)

to(*memory_format=torch.channels_last*)

Its signature is similar to `torch.Tensor.to()`, but only accepts floating point or complex `dtype`s. In addition, this method will only cast the floating point or complex parameters and buffers to `dtype` (if given). The integral parameters and buffers will be moved `device`, if that is given, but with dtypes unchanged. When `non_blocking` is set, it tries to convert/move asynchronously with respect to the host if possible, e.g., moving CPU Tensors with pinned memory to CUDA devices.

See below for examples.

Note: This method modifies the module in-place.

Parameters

- **device** (`torch.device`) – the desired device of the parameters and buffers in this module
- **dtype** (`torch.dtype`) – the desired floating point or complex dtype of the parameters and buffers in this module
- **tensor** (*torch.Tensor*) – Tensor whose dtype and device are the desired dtype and device for all parameters and buffers in this module
- **memory_format** (`torch.memory_format`) – the desired memory format for 4D parameters and buffers in this module (keyword only argument)

Returns self

Return type Module

Examples:

```
>>> # xdoctest: +IGNORE_WANT("non-deterministic")
>>> linear = nn.Linear(2, 2)
>>> linear.weight
Parameter containing:
tensor([[ 0.1913, -0.3420],
        [-0.5113, -0.2325]])
>>> linear.to(torch.double)
Linear(in_features=2, out_features=2, bias=True)
>>> linear.weight
Parameter containing:
tensor([[ 0.1913, -0.3420],
        [-0.5113, -0.2325]], dtype=torch.float64)
>>> # xdoctest: +REQUIRES(env:TORCH_DOCTEST_CUDA1)
>>> gpu1 = torch.device("cuda:1")
>>> linear.to(gpu1, dtype=torch.half, non_blocking=True)
Linear(in_features=2, out_features=2, bias=True)
>>> linear.weight
Parameter containing:
tensor([[ 0.1914, -0.3420],
        [-0.5112, -0.2324]], dtype=torch.float16, device='cuda:1')
>>> cpu = torch.device("cpu")
>>> linear.to(cpu)
```

(continues on next page)

(continued from previous page)

```
Linear(in_features=2, out_features=2, bias=True)
>>> linear.weight
Parameter containing:
tensor([[ 0.1914, -0.3420],
        [-0.5112, -0.2324]], dtype=torch.float16)

>>> linear = nn.Linear(2, 2, bias=None).to(torch.cdouble)
>>> linear.weight
Parameter containing:
tensor([[ 0.3741+0.j,  0.2382+0.j],
        [ 0.5593+0.j, -0.4443+0.j]], dtype=torch.complex128)
>>> linear(torch.ones(3, 2, dtype=torch.cdouble))
tensor([[0.6122+0.j, 0.1150+0.j],
        [0.6122+0.j, 0.1150+0.j],
        [0.6122+0.j, 0.1150+0.j]], dtype=torch.complex128)
```

to_empty(*, *device: Union[str, torch.device]*) → torch.nn.modules.module.T
Moves the parameters and buffers to the specified device without copying storage.

Parameters **device** (`torch.device`) – The desired device of the parameters and buffers in this module.

Returns self

Return type Module

train(*mode: bool = True*) → torch.nn.modules.module.T
Sets the module in training mode.

This has any effect only on certain modules. See documentations of particular modules for details of their behaviors in training/evaluation mode, if they are affected, e.g. `Dropout`, `BatchNorm`, etc.

Parameters **mode** (*bool*) – whether to set training mode (`True`) or evaluation mode (`False`). Default: `True`.

Returns self

Return type Module

training: bool

type(*dst_type: Union[torch.dtype, str]*) → torch.nn.modules.module.T
Casts all parameters and buffers to `dst_type`.

Note: This method modifies the module in-place.

Parameters **dst_type** (*type or string*) – the desired type

Returns self

Return type Module

xpu(*device: Optional[Union[int, torch.device]] = None*) → torch.nn.modules.module.T
Moves all model parameters and buffers to the XPU.

This also makes associated parameters and buffers different objects. So it should be called before constructing optimizer if the module will live on XPU while being optimized.

Note: This method modifies the module in-place.

Parameters **device** (*int, optional*) – if specified, all parameters will be copied to that device

Returns self

Return type Module

zero_grad(*set_to_none: bool = False*) → None
Sets gradients of all model parameters to zero. See similar function under `torch.optim.Optimizer` for more context.

Parameters **set_to_none** (*bool*) – instead of setting to zero, set the grads to None. See `torch.optim.Optimizer.zero_grad()` for details.

class TimeSeriesEncoder(*n_static_units: int*, *n_static_units_embedding: int*, *n_temporal_units: int*, *n_temporal_window: int*, *n_temporal_units_embedding: int*, *n_units_hidden: int = 200*, *n_layers_hidden: int = 2*, *nonlin: str = 'relu'*, *batch_norm: bool = False*, *dropout: float = 0.1*, *random_state: int = 0*, *mode: str = 'LSTM'*, *device: Any = device(type='cpu')*)
Bases: `torch.nn.modules.module.Module`

T_destination
alias of TypeVar('T_destination', bound=`Dict[str, Any]`)

add_module(*name: str*, *module: Optional[torch.nn.modules.module.Module]*) → None
Adds a child module to the current module.

The module can be accessed as an attribute using the given name.

Parameters

- **name** (*str*) – name of the child module. The child module can be accessed from this module using the given name
- **module** (*Module*) – child module to be added to the module.

apply(*fn: Callable[[torch.nn.modules.module.Module], None]*) → torch.nn.modules.module.T
Applies `fn` recursively to every submodule (as returned by `.children()`) as well as self. Typical use includes initializing the parameters of a model (see also nn-init-doc).

Parameters **fn** (`Module` -> None) – function to be applied to each submodule

Returns self

Return type Module

Example:

```
>>> @torch.no_grad()
>>> def init_weights(m):
>>>     print(m)
>>>     if type(m) == nn.Linear:
>>>         m.weight.fill_(1.0)
>>>         print(m.weight)
>>> net = nn.Sequential(nn.Linear(2, 2), nn.Linear(2, 2))
>>> net.apply(init_weights)
Linear(in_features=2, out_features=2, bias=True)
```

(continues on next page)

(continued from previous page)

```
Parameter containing:
tensor([[1., 1.],
        [1., 1.]], requires_grad=True)
Linear(in_features=2, out_features=2, bias=True)
Parameter containing:
tensor([[1., 1.],
        [1., 1.]], requires_grad=True)
Sequential(
  (0): Linear(in_features=2, out_features=2, bias=True)
  (1): Linear(in_features=2, out_features=2, bias=True)
)
```

bfloat16() → torch.nn.modules.module.T
Casts all floating point parameters and buffers to `bfloat16` datatype.

Note: This method modifies the module in-place.

Returns self

Return type Module

buffers(*recurse: bool = True*) → Iterator[torch.Tensor]
Returns an iterator over module buffers.

Parameters **recurse** (*bool*) – if True, then yields buffers of this module and all submodules. Otherwise, yields only buffers that are direct members of this module.

Yields *torch.Tensor* – module buffer

Example:

```
>>> # xdoctest: +SKIP("undefined vars")
>>> for buf in model.buffers():
>>>     print(type(buf), buf.size())
<class 'torch.Tensor'> (20L,)
<class 'torch.Tensor'> (20L, 1L, 5L, 5L)
```

children() → Iterator[torch.nn.modules.module.Module]
Returns an iterator over immediate children modules.

Yields *Module* – a child module

cpu() → torch.nn.modules.module.T
Moves all model parameters and buffers to the CPU.

Note: This method modifies the module in-place.

Returns self

Return type Module

cuda(*device: Optional[Union[int, torch.device]] = None*) → torch.nn.modules.module.T
Moves all model parameters and buffers to the GPU.

This also makes associated parameters and buffers different objects. So it should be called before constructing optimizer if the module will live on GPU while being optimized.

Note: This method modifies the module in-place.

Parameters **device** (*int, optional*) – if specified, all parameters will be copied to that device

Returns self

Return type Module

double() → torch.nn.modules.module.T
Casts all floating point parameters and buffers to double datatype.

Note: This method modifies the module in-place.

Returns self

Return type Module

dump_patches: bool = False

eval() → torch.nn.modules.module.T
Sets the module in evaluation mode.

This has any effect only on certain modules. See documentations of particular modules for details of their behaviors in training/evaluation mode, if they are affected, e.g. Dropout, BatchNorm, etc.

This is equivalent with self.train(False).

See locally-disable-grad-doc for a comparison between *.eval()* and several similar mechanisms that may be confused with it.

Returns self

Return type Module

extra_repr() → str
Set the extra representation of the module

To print customized extra information, you should re-implement this method in your own modules. Both single-line and multi-line strings are acceptable.

float() → torch.nn.modules.module.T
Casts all floating point parameters and buffers to float datatype.

Note: This method modifies the module in-place.

Returns self

Return type Module

forward(*static: torch.Tensor*, *temporal: torch.Tensor*, *observation_times: torch.Tensor*) →
Tuple[torch.Tensor, torch.Tensor, torch.Tensor, torch.Tensor, torch.Tensor, torch.Tensor]
Defines the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

`get_buffer`(*target: str*) → torch.Tensor
Returns the buffer given by `target` if it exists, otherwise throws an error.

See the docstring for `get_submodule` for a more detailed explanation of this method's functionality as well as how to correctly specify `target`.

Parameters `target` – The fully-qualified string name of the buffer to look for. (See `get_submodule` for how to specify a fully-qualified string.)

Returns The buffer referenced by `target`

Return type torch.Tensor

Raises `AttributeError` – If the target string references an invalid path or resolves to something that is not a buffer

`get_extra_state`() → Any
Returns any extra state to include in the module's state_dict. Implement this and a corresponding *set_extra_state()* for your module if you need to store extra state. This function is called when building the module's *state_dict()*.

Note that extra state should be pickleable to ensure working serialization of the state_dict. We only provide provide backwards compatibility guarantees for serializing Tensors; other objects may break backwards compatibility if their serialized pickled form changes.

Returns Any extra state to store in the module's state_dict

Return type object

`get_parameter`(*target: str*) → torch.nn.parameter.Parameter
Returns the parameter given by `target` if it exists, otherwise throws an error.

See the docstring for `get_submodule` for a more detailed explanation of this method's functionality as well as how to correctly specify `target`.

Parameters `target` – The fully-qualified string name of the Parameter to look for. (See `get_submodule` for how to specify a fully-qualified string.)

Returns The Parameter referenced by `target`

Return type torch.nn.Parameter

Raises `AttributeError` – If the target string references an invalid path or resolves to something that is not an `nn.Parameter`

`get_submodule`(*target: str*) → torch.nn.modules.module.Module
Returns the submodule given by `target` if it exists, otherwise throws an error.

For example, let's say you have an `nn.Module` `A` that looks like this:

```
A(
    (net_b): Module(
        (net_c): Module(
            (conv): Conv2d(16, 33, kernel_size=(3, 3), stride=(2, 2))
        )
```

(continues on next page)

(continued from previous page)

```
        (linear): Linear(in_features=100, out_features=200, bias=True)
    )
)
```

(The diagram shows an `nn.Module` `A`. `A` has a nested submodule `net_b`, which itself has two submodules `net_c` and `linear`. `net_c` then has a submodule `conv`.)

To check whether or not we have the `linear` submodule, we would call `get_submodule("net_b.linear")`. To check whether we have the `conv` submodule, we would call `get_submodule("net_b.net_c.conv")`.

The runtime of `get_submodule` is bounded by the degree of module nesting in `target`. A query against `named_modules` achieves the same result, but it is O(N) in the number of transitive modules. So, for a simple check to see if some submodule exists, `get_submodule` should always be used.

Parameters `target` – The fully-qualified string name of the submodule to look for. (See above example for how to specify a fully-qualified string.)

Returns The submodule referenced by `target`

Return type torch.nn.Module

Raises `AttributeError` – If the target string references an invalid path or resolves to something that is not an `nn.Module`

`half`() → torch.nn.modules.module.T

Casts all floating point parameters and buffers to `half` datatype.

Note: This method modifies the module in-place.

Returns self

Return type Module

`ipu`(*device: Optional[Union[int, torch.device]] = None*) → torch.nn.modules.module.T

Moves all model parameters and buffers to the IPU.

This also makes associated parameters and buffers different objects. So it should be called before constructing optimizer if the module will live on IPU while being optimized.

Note: This method modifies the module in-place.

Parameters `device` (*int, optional*) – if specified, all parameters will be copied to that device

Returns self

Return type Module

`load_state_dict`(*state_dict: Mapping[str, Any]*, *strict: bool = True*)

Copies parameters and buffers from *state_dict* into this module and its descendants. If `strict` is `True`, then the keys of *state_dict* must exactly match the keys returned by this module's `state_dict()` function.

Parameters

- **state_dict** (*dict*) – a dict containing parameters and persistent buffers.
- **strict** (*bool, optional*) – whether to strictly enforce that the keys in *state_dict* match the keys returned by this module's `state_dict()` function. Default: `True`

Returns

- **missing_keys** is a list of str containing the missing keys
- **unexpected_keys** is a list of str containing the unexpected keys

Return type `NamedTuple` with `missing_keys` and `unexpected_keys` fields

Note: If a parameter or buffer is registered as `None` and its corresponding key exists in *state_dict*, *load_state_dict()* will raise a `RuntimeError`.

---

modules() → Iterator[torch.nn.modules.module.Module]

Returns an iterator over all modules in the network.

Yields *Module* – a module in the network

Note: Duplicate modules are returned only once. In the following example, `l` will be returned only once.

Example:

```
>>> l = nn.Linear(2, 2)
>>> net = nn.Sequential(l, l)
>>> for idx, m in enumerate(net.modules()):
...     print(idx, '->', m)

0 -> Sequential(
  (0): Linear(in_features=2, out_features=2, bias=True)
  (1): Linear(in_features=2, out_features=2, bias=True)
)
1 -> Linear(in_features=2, out_features=2, bias=True)
```

named_buffers(*prefix: str = ''*, *recurse: bool = True*) → Iterator[Tuple[str, torch.Tensor]]

Returns an iterator over module buffers, yielding both the name of the buffer as well as the buffer itself.

Parameters

- **prefix** (*str*) – prefix to prepend to all buffer names.
- **recurse** (*bool*) – if True, then yields buffers of this module and all submodules. Otherwise, yields only buffers that are direct members of this module.

Yields *(str, torch.Tensor)* – Tuple containing the name and buffer

Example:

```
>>> # xdoctest: +SKIP("undefined vars")
>>> for name, buf in self.named_buffers():
>>>    if name in ['running_var']:
>>>        print(buf.size())
```

named_children() → Iterator[Tuple[str, torch.nn.modules.module.Module]]

Returns an iterator over immediate children modules, yielding both the name of the module as well as the module itself.

Yields *(str, Module)* – Tuple containing a name and child module

Example:

```
>>> # xdoctest: +SKIP("undefined vars")
>>> for name, module in model.named_children():
>>>     if name in ['conv4', 'conv5']:
>>>         print(module)
```

named_modules(*memo: Optional[Set[torch.nn.modules.module.Module]] = None*, *prefix: str = ''*, *remove_duplicate: bool = True*)

Returns an iterator over all modules in the network, yielding both the name of the module as well as the module itself.

Parameters

- **memo** – a memo to store the set of modules already added to the result
- **prefix** – a prefix that will be added to the name of the module
- **remove_duplicate** – whether to remove the duplicated module instances in the result or not

Yields *(str, Module)* – Tuple of name and module

Note: Duplicate modules are returned only once. In the following example, `l` will be returned only once.

Example:

```
>>> l = nn.Linear(2, 2)
>>> net = nn.Sequential(l, l)
>>> for idx, m in enumerate(net.named_modules()):
...     print(idx, '->', m)

0 -> ('', Sequential(
  (0): Linear(in_features=2, out_features=2, bias=True)
  (1): Linear(in_features=2, out_features=2, bias=True)
))
1 -> ('0', Linear(in_features=2, out_features=2, bias=True))
```

named_parameters(*prefix: str = ''*, *recurse: bool = True*) → Iterator[Tuple[str, torch.nn.parameter.Parameter]]

Returns an iterator over module parameters, yielding both the name of the parameter as well as the parameter itself.

Parameters

- **prefix** (*str*) – prefix to prepend to all parameter names.
- **recurse** (*bool*) – if True, then yields parameters of this module and all submodules. Otherwise, yields only parameters that are direct members of this module.

Yields *(str, Parameter)* – Tuple containing the name and parameter

Example:

```
>>> # xdoctest: +SKIP("undefined vars")
>>> for name, param in self.named_parameters():
```

(continues on next page)

(continued from previous page)

```
>>>     if name in ['bias']:
>>>         print(param.size())
```

parameters(*recurse: bool = True*) → Iterator[torch.nn.parameter.Parameter]
Returns an iterator over module parameters.

This is typically passed to an optimizer.

Parameters **recurse** (*bool*) – if True, then yields parameters of this module and all submodules. Otherwise, yields only parameters that are direct members of this module.

Yields *Parameter* – module parameter

Example:

```
>>> # xdoctest: +SKIP("undefined vars")
>>> for param in model.parameters():
>>>     print(type(param), param.size())
<class 'torch.Tensor'> (20L,)
<class 'torch.Tensor'> (20L, 1L, 5L, 5L)
```

register_backward_hook(*hook: Callable[[torch.nn.modules.module.Module, Union[Tuple[torch.Tensor, ...], torch.Tensor], Union[Tuple[torch.Tensor, ...], torch.Tensor]], Union[None, torch.Tensor]]*) → torch.utils.hooks.RemovableHandle
Registers a backward hook on the module.

This function is deprecated in favor of `register_full_backward_hook()` and the behavior of this function will change in future versions.

Returns a handle that can be used to remove the added hook by calling `handle.remove()`

Return type `torch.utils.hooks.RemovableHandle`

register_buffer(*name: str*, *tensor: Optional[torch.Tensor]*, *persistent: bool = True*) → None
Adds a buffer to the module.

This is typically used to register a buffer that should not to be considered a model parameter. For example, BatchNorm's `running_mean` is not a parameter, but is part of the module's state. Buffers, by default, are persistent and will be saved alongside parameters. This behavior can be changed by setting `persistent` to `False`. The only difference between a persistent buffer and a non-persistent buffer is that the latter will not be a part of this module's *state_dict*.

Buffers can be accessed as attributes using given names.

Parameters

- **name** (*str*) – name of the buffer. The buffer can be accessed from this module using the given name
- **tensor** (*Tensor or None*) – buffer to be registered. If `None`, then operations that run on buffers, such as *cuda*, are ignored. If `None`, the buffer is **not** included in the module's *state_dict*.
- **persistent** (*bool*) – whether the buffer is part of this module's *state_dict*.

Example:

```
>>> # xdoctest: +SKIP("undefined vars")
>>> self.register_buffer('running_mean', torch.zeros(num_features))
```

register_forward_hook(*hook: Callable[[...], None]*) → torch.utils.hooks.RemovableHandle
Registers a forward hook on the module.

The hook will be called every time after *forward()* has computed an output. It should have the following signature:

```
hook(module, input, output) -> None or modified output
```

The input contains only the positional arguments given to the module. Keyword arguments won't be passed to the hooks and only to the `forward`. The hook can modify the output. It can modify the input inplace but it will not have effect on forward since this is called after *forward()* is called.

Returns a handle that can be used to remove the added hook by calling `handle.remove()`

Return type `torch.utils.hooks.RemovableHandle`

register_forward_pre_hook(*hook: Callable[[...], None]*) → torch.utils.hooks.RemovableHandle
Registers a forward pre-hook on the module.

The hook will be called every time before *forward()* is invoked. It should have the following signature:

```
hook(module, input) -> None or modified input
```

The input contains only the positional arguments given to the module. Keyword arguments won't be passed to the hooks and only to the `forward`. The hook can modify the input. User can either return a tuple or a single modified value in the hook. We will wrap the value into a tuple if a single value is returned(unless that value is already a tuple).

Returns a handle that can be used to remove the added hook by calling `handle.remove()`

Return type `torch.utils.hooks.RemovableHandle`

register_full_backward_hook(*hook: Callable[[torch.nn.modules.module.Module, Union[Tuple[torch.Tensor, ...], torch.Tensor], Union[Tuple[torch.Tensor, ...], torch.Tensor]], Union[None, torch.Tensor]]*) → torch.utils.hooks.RemovableHandle
Registers a backward hook on the module.

The hook will be called every time the gradients with respect to a module are computed, i.e. the hook will execute if and only if the gradients with respect to module outputs are computed. The hook should have the following signature:

```
hook(module, grad_input, grad_output) -> tuple(Tensor) or None
```

The `grad_input` and `grad_output` are tuples that contain the gradients with respect to the inputs and outputs respectively. The hook should not modify its arguments, but it can optionally return a new gradient with respect to the input that will be used in place of `grad_input` in subsequent computations. `grad_input` will only correspond to the inputs given as positional arguments and all kwarg arguments are ignored. Entries in `grad_input` and `grad_output` will be `None` for all non-Tensor arguments.

For technical reasons, when this hook is applied to a Module, its forward function will receive a view of each Tensor passed to the Module. Similarly the caller will receive a view of each Tensor returned by the Module's forward function.

Warning: Modifying inputs or outputs inplace is not allowed when using backward hooks and will raise an error.

Returns a handle that can be used to remove the added hook by calling `handle.remove()`

Return type torch.utils.hooks.RemovableHandle

register_load_state_dict_post_hook(*hook*)
Registers a post hook to be run after module's load_state_dict is called.

It should have the following signature:: hook(module, incompatible_keys) -> None

The module argument is the current module that this hook is registered on, and the incompatible_keys argument is a NamedTuple consisting of attributes missing_keys and unexpected_keys. missing_keys is a list of str containing the missing keys and unexpected_keys is a list of str containing the unexpected keys.

The given incompatible_keys can be modified inplace if needed.

Note that the checks performed when calling *load_state_dict()* with strict=True are affected by modifications the hook makes to missing_keys or unexpected_keys, as expected. Additions to either set of keys will result in an error being thrown when strict=True, and clearning out both missing and unexpected keys will avoid an error.

Returns a handle that can be used to remove the added hook by calling handle.remove()

Return type torch.utils.hooks.RemovableHandle

register_module(*name: str*, *module: Optional[torch.nn.modules.module.Module]*) → None
Alias for *add_module()*.

register_parameter(*name: str*, *param: Optional[torch.nn.parameter.Parameter]*) → None
Adds a parameter to the module.

The parameter can be accessed as an attribute using given name.

Parameters

- **name** (*str*) – name of the parameter. The parameter can be accessed from this module using the given name
- **param** (*Parameter or None*) – parameter to be added to the module. If None, then operations that run on parameters, such as *cuda*, are ignored. If None, the parameter is **not** included in the module's *state_dict*.

requires_grad_(*requires_grad: bool = True*) → torch.nn.modules.module.T
Change if autograd should record operations on parameters in this module.

This method sets the parameters' requires_grad attributes in-place.

This method is helpful for freezing part of the module for finetuning or training parts of a model individually (e.g., GAN training).

See locally-disable-grad-doc for a comparison between *.requires_grad_()* and several similar mechanisms that may be confused with it.

Parameters **requires_grad** (*bool*) – whether autograd should record operations on parameters in this module. Default: True.

Returns self

Return type Module

set_extra_state(*state: Any*)
This function is called from *load_state_dict()* to handle any extra state found within the *state_dict*. Implement this function and a corresponding *get_extra_state()* for your module if you need to store extra state within its *state_dict*.

Parameters **state** (*dict*) – Extra state from the *state_dict*

share_memory() → torch.nn.modules.module.T
See torch.Tensor.share_memory_()

state_dict(**args*, *destination=None*, *prefix=''*, *keep_vars=False*)
Returns a dictionary containing references to the whole state of the module.

Both parameters and persistent buffers (e.g. running averages) are included. Keys are corresponding parameter and buffer names. Parameters and buffers set to None are not included.

Note: The returned object is a shallow copy. It contains references to the module's parameters and buffers.

Warning: Currently state_dict() also accepts positional arguments for destination, prefix and keep_vars in order. However, this is being deprecated and keyword arguments will be enforced in future releases.

Warning: Please avoid the use of argument destination as it is not designed for end-users.

Parameters

- **destination** (*dict, optional*) – If provided, the state of module will be updated into the dict and the same object is returned. Otherwise, an OrderedDict will be created and returned. Default: None.
- **prefix** (*str, optional*) – a prefix added to parameter and buffer names to compose the keys in state_dict. Default: ''.
- **keep_vars** (*bool, optional*) – by default the Tensor s returned in the state dict are detached from autograd. If it's set to True, detaching will not be performed. Default: False.

Returns a dictionary containing a whole state of the module

Return type dict

Example:

```
>>> # xdoctest: +SKIP("undefined vars")
>>> module.state_dict().keys()
['bias', 'weight']
```

to(**args*, ***kwargs*)
Moves and/or casts the parameters and buffers.

This can be called as

to(*device=None*, *dtype=None*, *non_blocking=False*)

to(*dtype*, *non_blocking=False*)

to(*tensor*, *non_blocking=False*)

to(*memory_format=torch.channels_last*)

Its signature is similar to torch.Tensor.to(), but only accepts floating point or complex dtypes. In addition, this method will only cast the floating point or complex parameters and buffers to dtype (if given). The integral parameters and buffers will be moved device, if that is given, but with dtypes unchanged.

When non_blocking is set, it tries to convert/move asynchronously with respect to the host if possible, e.g., moving CPU Tensors with pinned memory to CUDA devices.

See below for examples.

Note: This method modifies the module in-place.

Parameters

- **device** (torch.device) – the desired device of the parameters and buffers in this module
- **dtype** (torch.dtype) – the desired floating point or complex dtype of the parameters and buffers in this module
- **tensor** (*torch.Tensor*) – Tensor whose dtype and device are the desired dtype and device for all parameters and buffers in this module
- **memory_format** (torch.memory_format) – the desired memory format for 4D parameters and buffers in this module (keyword only argument)

Returns self

Return type Module

Examples:

```
>>> # xdoctest: +IGNORE_WANT("non-deterministic")
>>> linear = nn.Linear(2, 2)
>>> linear.weight
Parameter containing:
tensor([[ 0.1913, -0.3420],
        [-0.5113, -0.2325]])
>>> linear.to(torch.double)
Linear(in_features=2, out_features=2, bias=True)
>>> linear.weight
Parameter containing:
tensor([[ 0.1913, -0.3420],
        [-0.5113, -0.2325]], dtype=torch.float64)
>>> # xdoctest: +REQUIRES(env:TORCH_DOCTEST_CUDA1)
>>> gpu1 = torch.device("cuda:1")
>>> linear.to(gpu1, dtype=torch.half, non_blocking=True)
Linear(in_features=2, out_features=2, bias=True)
>>> linear.weight
Parameter containing:
tensor([[ 0.1914, -0.3420],
        [-0.5112, -0.2324]], dtype=torch.float16, device='cuda:1')
>>> cpu = torch.device("cpu")
>>> linear.to(cpu)
Linear(in_features=2, out_features=2, bias=True)
>>> linear.weight
Parameter containing:
tensor([[ 0.1914, -0.3420],
        [-0.5112, -0.2324]], dtype=torch.float16)
```

(continues on next page)

(continued from previous page)

```
>>> linear = nn.Linear(2, 2, bias=None).to(torch.cdouble)
>>> linear.weight
Parameter containing:
tensor([[ 0.3741+0.j,  0.2382+0.j],
        [ 0.5593+0.j, -0.4443+0.j]], dtype=torch.complex128)
>>> linear(torch.ones(3, 2, dtype=torch.cdouble))
tensor([[0.6122+0.j, 0.1150+0.j],
        [0.6122+0.j, 0.1150+0.j],
        [0.6122+0.j, 0.1150+0.j]], dtype=torch.complex128)
```

to_empty(*, *device: Union[str, torch.device]*) → torch.nn.modules.module.T
Moves the parameters and buffers to the specified device without copying storage.

Parameters **device** (torch.device) – The desired device of the parameters and buffers in this module.

Returns self

Return type Module

train(*mode: bool = True*) → torch.nn.modules.module.T
Sets the module in training mode.

This has any effect only on certain modules. See documentations of particular modules for details of their behaviors in training/evaluation mode, if they are affected, e.g. Dropout, BatchNorm, etc.

Parameters **mode** (*bool*) – whether to set training mode (True) or evaluation mode (False). Default: True.

Returns self

Return type Module

training: bool

type(*dst_type: Union[torch.dtype, str]*) → torch.nn.modules.module.T
Casts all parameters and buffers to dst_type.

Note: This method modifies the module in-place.

Parameters **dst_type** (*type or string*) – the desired type

Returns self

Return type Module

xpu(*device: Optional[Union[int, torch.device]] = None*) → torch.nn.modules.module.T
Moves all model parameters and buffers to the XPU.

This also makes associated parameters and buffers different objects. So it should be called before constructing optimizer if the module will live on XPU while being optimized.

Note: This method modifies the module in-place.

Parameters **device** (*int, optional*) – if specified, all parameters will be copied to that device

Returns self

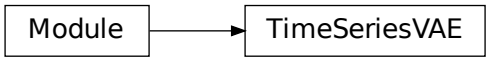
Return type Module

zero_grad(*set_to_none: bool = False*) → None
Sets gradients of all model parameters to zero. See similar function under `torch.optim.Optimizer` for more context.

Parameters **set_to_none** (*bool*) – instead of setting to zero, set the grads to None. See `torch.optim.Optimizer.zero_grad()` for details.

class TimeSeriesVAE(*n_static_units: int*, *n_static_units_embedding: int*, *n_temporal_units: int*, *n_temporal_window: int*, *n_temporal_units_embedding: int*, *batch_size: int = 100*, *n_iter: int = 500*, *n_iter_print: int = 10*, *random_state: int = 0*, *clipping_value: int = 1*, *lr: float = 0.0002*, *weight_decay: float = 0.001*, *loss_factor: int = 2*, *decoder_n_layers_hidden: int = 2*, *decoder_n_units_hidden: int = 250*, *decoder_nonlin: str = 'leaky_relu'*, *decoder_static_nonlin_out: Optional[List[Tuple[str, int]]] = None*, *decoder_temporal_nonlin_out: Optional[List[Tuple[str, int]]] = None*, *decoder_batch_norm: bool = False*, *decoder_dropout: float = 0*, *decoder_residual: bool = True*, *decoder_mode: str = 'LSTM'*, *encoder_n_layers_hidden: int = 3*, *encoder_n_units_hidden: int = 300*, *encoder_nonlin: str = 'leaky_relu'*, *encoder_batch_norm: bool = False*, *encoder_dropout: float = 0.1*, *encoder_mode: str = 'LSTM'*, *device: Any = device(type='cpu')*)

Bases: `torch.nn.modules.module.Module`



Basic Time-Series Variational AutoEncoder

Parameters

- **n_static_units** – int Number of input static units.
- **n_static_units_embedding** – int. Number of latent static units.
- **n_temporal_units** – int. Number of input temporal units.
- **n_temporal_window** – int The length of time series sequences.
- **n_temporal_units_embedding** – int. Number of latent temporal units.
- **batch_size** – int. Default = 100 Batch size.
- **n_iter** – int. Default = 500 Number of epochs
- **n_iter_print** – int = 10. Frequency of printing the training loss.
- **random_state** – int = 0 Random seed
- **clipping_value** – int = 1 Gradients clipping value. Zero disables the feature
- **lr** – float = 2e-4, Learning rate
- **weight_decay** – float = 1e-3, l2 (ridge) penalty for the weights.
- **loss_factor** – int. Default = 2 Reconstruction loss weight.

- **decoder_n_layers_hidden** – int. Default = 2 Number of hidden layer in the decoder
- **decoder_n_units_hidden** – int. Decoder = 250. Number of hidden units in the decoder
- **decoder_nonlin** – str. Decoder = "leaky_relu" Activation for the hidden layers. Can be relu, elu, leaky_relu, selu.
- **decoder_static_nonlin_out** – Optional[List[Tuple[str, int]]] = None (Optional) Activations to use in the output layer of the decoder for static features.
- **decoder_temporal_nonlin_out** – Optional[List[Tuple[str, int]]] = None, (Optional) Activations to use in the output layer of the decoder for temporal features.
- **decoder_batch_norm** – bool. Default = False Decoder batch norm
- **decoder_dropout** – float. Default = 0 Decoder dropout
- **decoder_residual** – bool. Default = True Use residual connections in the decoder
- **decoder_mode** – str. Default = "LSTM" Core neural net architecture for the decoder. Available models:
  - "LSTM"
  - "GRU"
  - "RNN"
  - "Transformer"
  - "MLSTM_FCN"
  - "TCN"
  - "InceptionTime"
  - "InceptionTimePlus"
  - "XceptionTime"
  - "ResCNN"
  - "OmniScaleCNN"
  - "XCM"
- **encoder_n_layers_hidden** – int. Default = 3 Number of hidden layers in the encoder
- **300** (*encoder_n_units_hidden. Default int =*) – Number of hidden units in the encoder
- **encoder_nonlin** – str. Default = "leaky_relu" Activations for the hidden layers in the encoder.
- **encoder_batch_norm** – bool. Default = False, Encoder batch norm.
- **encoder_dropout** – float. Default = 0.1 Encoder dropout.
- **encoder_mode** – str. Default = "LSTM" Core neural net architecture for the encoder. Available models:
  - "LSTM"
  - "GRU"
  - "RNN"
  - "Transformer"

- "MLSTM_FCN"
- "TCN"
- "InceptionTime"
- "InceptionTimePlus"
- "XceptionTime"
- "ResCNN"
- "OmniScaleCNN"
- "XCM"
- "Transformer"

• **device** – PyTorch device: cpu/cuda.

T_destination
alias of TypeVar('T_destination', bound=Dict[str, Any])

add_module(*name: str*, *module: Optional[torch.nn.modules.module.Module]*) → None
Adds a child module to the current module.

The module can be accessed as an attribute using the given name.

Parameters

• **name** (*str*) – name of the child module. The child module can be accessed from this module using the given name

• **module** (*Module*) – child module to be added to the module.

apply(*fn: Callable[[torch.nn.modules.module.Module], None]*) → torch.nn.modules.module.T
Applies `fn` recursively to every submodule (as returned by `.children()`) as well as self. Typical use includes initializing the parameters of a model (see also nn-init-doc).

Parameters **fn** (Module -> None) – function to be applied to each submodule

Returns self

Return type Module

Example:

```
>>> @torch.no_grad()
>>> def init_weights(m):
>>>     print(m)
>>>     if type(m) == nn.Linear:
>>>         m.weight.fill_(1.0)
>>>         print(m.weight)
>>> net = nn.Sequential(nn.Linear(2, 2), nn.Linear(2, 2))
>>> net.apply(init_weights)
Linear(in_features=2, out_features=2, bias=True)
Parameter containing:
tensor([[1., 1.],
        [1., 1.]], requires_grad=True)
Linear(in_features=2, out_features=2, bias=True)
Parameter containing:
tensor([[1., 1.],
        [1., 1.]], requires_grad=True)
```

(continues on next page)

(continued from previous page)

```
Sequential(
  (0): Linear(in_features=2, out_features=2, bias=True)
  (1): Linear(in_features=2, out_features=2, bias=True)
)
```

bfloat16() → torch.nn.modules.module.T
Casts all floating point parameters and buffers to `bfloat16` datatype.

Note: This method modifies the module in-place.

Returns self

Return type Module

buffers(*recurse: bool = True*) → Iterator[torch.Tensor]
Returns an iterator over module buffers.

Parameters **recurse** (*bool*) – if True, then yields buffers of this module and all submodules. Otherwise, yields only buffers that are direct members of this module.

Yields *torch.Tensor* – module buffer

Example:

```
>>> # xdoctest: +SKIP("undefined vars")
>>> for buf in model.buffers():
>>>     print(type(buf), buf.size())
<class 'torch.Tensor'> (20L,)
<class 'torch.Tensor'> (20L, 1L, 5L, 5L)
```

children() → Iterator[torch.nn.modules.module.Module]
Returns an iterator over immediate children modules.

Yields *Module* – a child module

cpu() → torch.nn.modules.module.T
Moves all model parameters and buffers to the CPU.

Note: This method modifies the module in-place.

Returns self

Return type Module

cuda(*device: Optional[Union[int, torch.device]] = None*) → torch.nn.modules.module.T
Moves all model parameters and buffers to the GPU.

This also makes associated parameters and buffers different objects. So it should be called before constructing optimizer if the module will live on GPU while being optimized.

Note: This method modifies the module in-place.

Parameters `device` (*int, optional*) – if specified, all parameters will be copied to that device

Returns self

Return type Module

`double`() → torch.nn.modules.module.T
Casts all floating point parameters and buffers to `double` datatype.

Note: This method modifies the module in-place.

Returns self

Return type Module

`dump_patches: bool = False`

`eval`() → torch.nn.modules.module.T
Sets the module in evaluation mode.

This has any effect only on certain modules. See documentations of particular modules for details of their behaviors in training/evaluation mode, if they are affected, e.g. `Dropout`, `BatchNorm`, etc.

This is equivalent with `self.train(False)`.

See locally-disable-grad-doc for a comparison between *.eval()* and several similar mechanisms that may be confused with it.

Returns self

Return type Module

`extra_repr`() → str
Set the extra representation of the module

To print customized extra information, you should re-implement this method in your own modules. Both single-line and multi-line strings are acceptable.

`fit`(*static: numpy.ndarray*, *temporal: numpy.ndarray*, *observation_times: numpy.ndarray*) → Any

`float`() → torch.nn.modules.module.T
Casts all floating point parameters and buffers to `float` datatype.

Note: This method modifies the module in-place.

Returns self

Return type Module

`forward`(**input: Any*) → None
Defines the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

`generate`(*samples: int*) → numpy.ndarray

`get_buffer`(*target: str*) → torch.Tensor

Returns the buffer given by `target` if it exists, otherwise throws an error.

See the docstring for `get_submodule` for a more detailed explanation of this method's functionality as well as how to correctly specify `target`.

Parameters `target` – The fully-qualified string name of the buffer to look for. (See `get_submodule` for how to specify a fully-qualified string.)

Returns The buffer referenced by `target`

Return type torch.Tensor

Raises `AttributeError` – If the target string references an invalid path or resolves to something that is not a buffer

`get_extra_state`() → Any

Returns any extra state to include in the module's state_dict. Implement this and a corresponding *set_extra_state()* for your module if you need to store extra state. This function is called when building the module's *state_dict()*.

Note that extra state should be pickleable to ensure working serialization of the state_dict. We only provide provide backwards compatibility guarantees for serializing Tensors; other objects may break backwards compatibility if their serialized pickled form changes.

Returns Any extra state to store in the module's state_dict

Return type object

`get_parameter`(*target: str*) → torch.nn.parameter.Parameter

Returns the parameter given by `target` if it exists, otherwise throws an error.

See the docstring for `get_submodule` for a more detailed explanation of this method's functionality as well as how to correctly specify `target`.

Parameters `target` – The fully-qualified string name of the Parameter to look for. (See `get_submodule` for how to specify a fully-qualified string.)

Returns The Parameter referenced by `target`

Return type torch.nn.Parameter

Raises `AttributeError` – If the target string references an invalid path or resolves to something that is not an `nn.Parameter`

`get_submodule`(*target: str*) → torch.nn.modules.module.Module

Returns the submodule given by `target` if it exists, otherwise throws an error.

For example, let's say you have an `nn.Module` `A` that looks like this:

```
A(
    (net_b): Module(
        (net_c): Module(
            (conv): Conv2d(16, 33, kernel_size=(3, 3), stride=(2, 2))
        )
```

(continues on next page)

(continued from previous page)

```
        (linear): Linear(in_features=100, out_features=200, bias=True)
    )
)
```

(The diagram shows an `nn.Module` `A`. `A` has a nested submodule `net_b`, which itself has two submodules `net_c` and `linear`. `net_c` then has a submodule `conv`.)

To check whether or not we have the `linear` submodule, we would call `get_submodule("net_b.linear")`. To check whether we have the `conv` submodule, we would call `get_submodule("net_b.net_c.conv")`.

The runtime of `get_submodule` is bounded by the degree of module nesting in `target`. A query against `named_modules` achieves the same result, but it is O(N) in the number of transitive modules. So, for a simple check to see if some submodule exists, `get_submodule` should always be used.

Parameters `target` – The fully-qualified string name of the submodule to look for. (See above example for how to specify a fully-qualified string.)

Returns The submodule referenced by `target`

Return type torch.nn.Module

Raises `AttributeError` – If the target string references an invalid path or resolves to something that is not an `nn.Module`

`half`() → torch.nn.modules.module.T

Casts all floating point parameters and buffers to `half` datatype.

Note: This method modifies the module in-place.

Returns self

Return type Module

`ipu`(*device: Optional[Union[int, torch.device]] = None*) → torch.nn.modules.module.T

Moves all model parameters and buffers to the IPU.

This also makes associated parameters and buffers different objects. So it should be called before constructing optimizer if the module will live on IPU while being optimized.

Note: This method modifies the module in-place.

Parameters `device` (*int, optional*) – if specified, all parameters will be copied to that device

Returns self

Return type Module

`load_state_dict`(*state_dict: Mapping[str, Any]*, *strict: bool = True*)

Copies parameters and buffers from *state_dict* into this module and its descendants. If `strict` is `True`, then the keys of *state_dict* must exactly match the keys returned by this module's `state_dict()` function.

Parameters

- **state_dict** (*dict*) – a dict containing parameters and persistent buffers.
- **strict** (*bool, optional*) – whether to strictly enforce that the keys in *state_dict* match the keys returned by this module's `state_dict()` function. Default: `True`

Returns

- **missing_keys** is a list of str containing the missing keys
- **unexpected_keys** is a list of str containing the unexpected keys

Return type `NamedTuple` with `missing_keys` and `unexpected_keys` fields

Note: If a parameter or buffer is registered as `None` and its corresponding key exists in *state_dict*, *load_state_dict()* will raise a `RuntimeError`.

---

modules() → Iterator[torch.nn.modules.module.Module]
Returns an iterator over all modules in the network.

Yields *Module* – a module in the network

Note: Duplicate modules are returned only once. In the following example, `l` will be returned only once.

Example:

```
>>> l = nn.Linear(2, 2)
>>> net = nn.Sequential(l, l)
>>> for idx, m in enumerate(net.modules()):
...     print(idx, '->', m)

0 -> Sequential(
  (0): Linear(in_features=2, out_features=2, bias=True)
  (1): Linear(in_features=2, out_features=2, bias=True)
)
1 -> Linear(in_features=2, out_features=2, bias=True)
```

named_buffers(*prefix: str = ''*, *recurse: bool = True*) → Iterator[Tuple[str, torch.Tensor]]
Returns an iterator over module buffers, yielding both the name of the buffer as well as the buffer itself.

Parameters

- **prefix** (*str*) – prefix to prepend to all buffer names.
- **recurse** (*bool*) – if True, then yields buffers of this module and all submodules. Otherwise, yields only buffers that are direct members of this module.

Yields *(str, torch.Tensor)* – Tuple containing the name and buffer

Example:

```
>>> # xdoctest: +SKIP("undefined vars")
>>> for name, buf in self.named_buffers():
>>>    if name in ['running_var']:
>>>        print(buf.size())
```

named_children() → Iterator[Tuple[str, torch.nn.modules.module.Module]]
Returns an iterator over immediate children modules, yielding both the name of the module as well as the module itself.

Yields *(str, Module)* – Tuple containing a name and child module

Example:

```
>>> # xdoctest: +SKIP("undefined vars")
>>> for name, module in model.named_children():
>>>     if name in ['conv4', 'conv5']:
>>>         print(module)
```

named_modules(*memo: Optional[Set[torch.nn.modules.module.Module]] = None*, *prefix: str = ''*, *remove_duplicate: bool = True*)

Returns an iterator over all modules in the network, yielding both the name of the module as well as the module itself.

Parameters

- **memo** – a memo to store the set of modules already added to the result
- **prefix** – a prefix that will be added to the name of the module
- **remove_duplicate** – whether to remove the duplicated module instances in the result or not

Yields *(str, Module)* – Tuple of name and module

Note: Duplicate modules are returned only once. In the following example, `l` will be returned only once.

Example:

```
>>> l = nn.Linear(2, 2)
>>> net = nn.Sequential(l, l)
>>> for idx, m in enumerate(net.named_modules()):
...     print(idx, '->', m)

0 -> ('', Sequential(
  (0): Linear(in_features=2, out_features=2, bias=True)
  (1): Linear(in_features=2, out_features=2, bias=True)
))
1 -> ('0', Linear(in_features=2, out_features=2, bias=True))
```

named_parameters(*prefix: str = ''*, *recurse: bool = True*) → Iterator[Tuple[str, torch.nn.parameter.Parameter]]

Returns an iterator over module parameters, yielding both the name of the parameter as well as the parameter itself.

Parameters

- **prefix** (*str*) – prefix to prepend to all parameter names.
- **recurse** (*bool*) – if True, then yields parameters of this module and all submodules. Otherwise, yields only parameters that are direct members of this module.

Yields *(str, Parameter)* – Tuple containing the name and parameter

Example:

```
>>> # xdoctest: +SKIP("undefined vars")
>>> for name, param in self.named_parameters():
```

(continues on next page)

(continued from previous page)

```
>>>     if name in ['bias']:
>>>         print(param.size())
```

parameters(*recurse: bool = True*) → Iterator[torch.nn.parameter.Parameter]
Returns an iterator over module parameters.

This is typically passed to an optimizer.

Parameters **recurse** (*bool*) – if True, then yields parameters of this module and all submodules. Otherwise, yields only parameters that are direct members of this module.

Yields *Parameter* – module parameter

Example:

```
>>> # xdoctest: +SKIP("undefined vars")
>>> for param in model.parameters():
>>>     print(type(param), param.size())
<class 'torch.Tensor'> (20L,)
<class 'torch.Tensor'> (20L, 1L, 5L, 5L)
```

register_backward_hook(*hook: Callable[[torch.nn.modules.module.Module, Union[Tuple[torch.Tensor, ...], torch.Tensor], Union[Tuple[torch.Tensor, ...], torch.Tensor]], Union[None, torch.Tensor]]*) → torch.utils.hooks.RemovableHandle
Registers a backward hook on the module.

This function is deprecated in favor of `register_full_backward_hook()` and the behavior of this function will change in future versions.

Returns a handle that can be used to remove the added hook by calling `handle.remove()`

Return type `torch.utils.hooks.RemovableHandle`

register_buffer(*name: str*, *tensor: Optional[torch.Tensor]*, *persistent: bool = True*) → None
Adds a buffer to the module.

This is typically used to register a buffer that should not to be considered a model parameter. For example, BatchNorm's `running_mean` is not a parameter, but is part of the module's state. Buffers, by default, are persistent and will be saved alongside parameters. This behavior can be changed by setting `persistent` to `False`. The only difference between a persistent buffer and a non-persistent buffer is that the latter will not be a part of this module's *state_dict*.

Buffers can be accessed as attributes using given names.

Parameters

- **name** (*str*) – name of the buffer. The buffer can be accessed from this module using the given name
- **tensor** (*Tensor or None*) – buffer to be registered. If `None`, then operations that run on buffers, such as *cuda*, are ignored. If `None`, the buffer is **not** included in the module's *state_dict*.
- **persistent** (*bool*) – whether the buffer is part of this module's *state_dict*.

Example:

```
>>> # xdoctest: +SKIP("undefined vars")
>>> self.register_buffer('running_mean', torch.zeros(num_features))
```

register_forward_hook(*hook: Callable[[...], None]*) → torch.utils.hooks.RemovableHandle
Registers a forward hook on the module.

The hook will be called every time after *forward()* has computed an output. It should have the following signature:

```
hook(module, input, output) -> None or modified output
```

The input contains only the positional arguments given to the module. Keyword arguments won't be passed to the hooks and only to the `forward`. The hook can modify the output. It can modify the input inplace but it will not have effect on forward since this is called after *forward()* is called.

Returns a handle that can be used to remove the added hook by calling `handle.remove()`

Return type `torch.utils.hooks.RemovableHandle`

register_forward_pre_hook(*hook: Callable[[...], None]*) → torch.utils.hooks.RemovableHandle
Registers a forward pre-hook on the module.

The hook will be called every time before *forward()* is invoked. It should have the following signature:

```
hook(module, input) -> None or modified input
```

The input contains only the positional arguments given to the module. Keyword arguments won't be passed to the hooks and only to the `forward`. The hook can modify the input. User can either return a tuple or a single modified value in the hook. We will wrap the value into a tuple if a single value is returned(unless that value is already a tuple).

Returns a handle that can be used to remove the added hook by calling `handle.remove()`

Return type `torch.utils.hooks.RemovableHandle`

register_full_backward_hook(*hook: Callable[[torch.nn.modules.module.Module, Union[Tuple[torch.Tensor, ...], torch.Tensor], Union[Tuple[torch.Tensor, ...], torch.Tensor]], Union[None, torch.Tensor]]*) → torch.utils.hooks.RemovableHandle
Registers a backward hook on the module.

The hook will be called every time the gradients with respect to a module are computed, i.e. the hook will execute if and only if the gradients with respect to module outputs are computed. The hook should have the following signature:

```
hook(module, grad_input, grad_output) -> tuple(Tensor) or None
```

The `grad_input` and `grad_output` are tuples that contain the gradients with respect to the inputs and outputs respectively. The hook should not modify its arguments, but it can optionally return a new gradient with respect to the input that will be used in place of `grad_input` in subsequent computations. `grad_input` will only correspond to the inputs given as positional arguments and all kwarg arguments are ignored. Entries in `grad_input` and `grad_output` will be `None` for all non-Tensor arguments.

For technical reasons, when this hook is applied to a Module, its forward function will receive a view of each Tensor passed to the Module. Similarly the caller will receive a view of each Tensor returned by the Module's forward function.

Warning: Modifying inputs or outputs inplace is not allowed when using backward hooks and will raise an error.

Returns a handle that can be used to remove the added hook by calling `handle.remove()`

Return type torch.utils.hooks.RemovableHandle

register_load_state_dict_post_hook(*hook*)
Registers a post hook to be run after module's load_state_dict is called.

It should have the following signature:: hook(module, incompatible_keys) -> None

The module argument is the current module that this hook is registered on, and the incompatible_keys argument is a NamedTuple consisting of attributes missing_keys and unexpected_keys. missing_keys is a list of str containing the missing keys and unexpected_keys is a list of str containing the unexpected keys.

The given incompatible_keys can be modified inplace if needed.

Note that the checks performed when calling *load_state_dict()* with strict=True are affected by modifications the hook makes to missing_keys or unexpected_keys, as expected. Additions to either set of keys will result in an error being thrown when strict=True, and clearning out both missing and unexpected keys will avoid an error.

Returns a handle that can be used to remove the added hook by calling handle.remove()

Return type torch.utils.hooks.RemovableHandle

register_module(*name: str*, *module: Optional[torch.nn.modules.module.Module]*) → None
Alias for *add_module()*.

register_parameter(*name: str*, *param: Optional[torch.nn.parameter.Parameter]*) → None
Adds a parameter to the module.

The parameter can be accessed as an attribute using given name.

Parameters

- **name** (*str*) – name of the parameter. The parameter can be accessed from this module using the given name
- **param** (*Parameter or None*) – parameter to be added to the module. If None, then operations that run on parameters, such as *cuda*, are ignored. If None, the parameter is **not** included in the module's *state_dict*.

requires_grad_(*requires_grad: bool = True*) → torch.nn.modules.module.T
Change if autograd should record operations on parameters in this module.

This method sets the parameters' requires_grad attributes in-place.

This method is helpful for freezing part of the module for finetuning or training parts of a model individually (e.g., GAN training).

See locally-disable-grad-doc for a comparison between *.requires_grad_()* and several similar mechanisms that may be confused with it.

Parameters **requires_grad** (*bool*) – whether autograd should record operations on parameters in this module. Default: True.

Returns self

Return type Module

set_extra_state(*state: Any*)
This function is called from *load_state_dict()* to handle any extra state found within the *state_dict*. Implement this function and a corresponding *get_extra_state()* for your module if you need to store extra state within its *state_dict*.

Parameters **state** (*dict*) – Extra state from the *state_dict*

share_memory() → torch.nn.modules.module.T
See `torch.Tensor.share_memory_()`

state_dict(**args*, *destination=None*, *prefix=''*, *keep_vars=False*)
Returns a dictionary containing references to the whole state of the module.

Both parameters and persistent buffers (e.g. running averages) are included. Keys are corresponding parameter and buffer names. Parameters and buffers set to `None` are not included.

Note: The returned object is a shallow copy. It contains references to the module's parameters and buffers.

Warning: Currently `state_dict()` also accepts positional arguments for `destination`, `prefix` and `keep_vars` in order. However, this is being deprecated and keyword arguments will be enforced in future releases.

Warning: Please avoid the use of argument `destination` as it is not designed for end-users.

Parameters

- **destination** (*dict, optional*) – If provided, the state of module will be updated into the dict and the same object is returned. Otherwise, an `OrderedDict` will be created and returned. Default: `None`.
- **prefix** (*str, optional*) – a prefix added to parameter and buffer names to compose the keys in state_dict. Default: `''`.
- **keep_vars** (*bool, optional*) – by default the `Tensor` s returned in the state dict are detached from autograd. If it's set to `True`, detaching will not be performed. Default: `False`.

Returns a dictionary containing a whole state of the module

Return type dict

Example:

```
>>> # xdoctest: +SKIP("undefined vars")
>>> module.state_dict().keys()
['bias', 'weight']
```

to(**args*, ***kwargs*)
Moves and/or casts the parameters and buffers.

This can be called as

to(*device=None*, *dtype=None*, *non_blocking=False*)

to(*dtype*, *non_blocking=False*)

to(*tensor*, *non_blocking=False*)

to(*memory_format=torch.channels_last*)

Its signature is similar to `torch.Tensor.to()`, but only accepts floating point or complex `dtypes`. In addition, this method will only cast the floating point or complex parameters and buffers to `dtype` (if given). The integral parameters and buffers will be moved `device`, if that is given, but with dtypes unchanged.

When non_blocking is set, it tries to convert/move asynchronously with respect to the host if possible, e.g., moving CPU Tensors with pinned memory to CUDA devices.

See below for examples.

Note: This method modifies the module in-place.

Parameters

- **device** (torch.device) – the desired device of the parameters and buffers in this module
- **dtype** (torch.dtype) – the desired floating point or complex dtype of the parameters and buffers in this module
- **tensor** (*torch.Tensor*) – Tensor whose dtype and device are the desired dtype and device for all parameters and buffers in this module
- **memory_format** (torch.memory_format) – the desired memory format for 4D parameters and buffers in this module (keyword only argument)

Returns self

Return type Module

Examples:

```
>>> # xdoctest: +IGNORE_WANT("non-deterministic")
>>> linear = nn.Linear(2, 2)
>>> linear.weight
Parameter containing:
tensor([[ 0.1913, -0.3420],
        [-0.5113, -0.2325]])
>>> linear.to(torch.double)
Linear(in_features=2, out_features=2, bias=True)
>>> linear.weight
Parameter containing:
tensor([[ 0.1913, -0.3420],
        [-0.5113, -0.2325]], dtype=torch.float64)
>>> # xdoctest: +REQUIRES(env:TORCH_DOCTEST_CUDA1)
>>> gpu1 = torch.device("cuda:1")
>>> linear.to(gpu1, dtype=torch.half, non_blocking=True)
Linear(in_features=2, out_features=2, bias=True)
>>> linear.weight
Parameter containing:
tensor([[ 0.1914, -0.3420],
        [-0.5112, -0.2324]], dtype=torch.float16, device='cuda:1')
>>> cpu = torch.device("cpu")
>>> linear.to(cpu)
Linear(in_features=2, out_features=2, bias=True)
>>> linear.weight
Parameter containing:
tensor([[ 0.1914, -0.3420],
        [-0.5112, -0.2324]], dtype=torch.float16)
```

(continues on next page)

(continued from previous page)

```
>>> linear = nn.Linear(2, 2, bias=None).to(torch.cdouble)
>>> linear.weight
Parameter containing:
tensor([[ 0.3741+0.j,  0.2382+0.j],
        [ 0.5593+0.j, -0.4443+0.j]], dtype=torch.complex128)
>>> linear(torch.ones(3, 2, dtype=torch.cdouble))
tensor([[0.6122+0.j, 0.1150+0.j],
        [0.6122+0.j, 0.1150+0.j],
        [0.6122+0.j, 0.1150+0.j]], dtype=torch.complex128)
```

to_empty(*, *device: Union[str, torch.device]*) → torch.nn.modules.module.T
Moves the parameters and buffers to the specified device without copying storage.

Parameters **device** (torch.device) – The desired device of the parameters and buffers in this module.

Returns self

Return type Module

train(*mode: bool = True*) → torch.nn.modules.module.T
Sets the module in training mode.

This has any effect only on certain modules. See documentations of particular modules for details of their behaviors in training/evaluation mode, if they are affected, e.g. Dropout, BatchNorm, etc.

Parameters **mode** (*bool*) – whether to set training mode (True) or evaluation mode (False). Default: True.

Returns self

Return type Module

training: bool

type(*dst_type: Union[torch.dtype, str]*) → torch.nn.modules.module.T
Casts all parameters and buffers to dst_type.

Note: This method modifies the module in-place.

Parameters **dst_type** (*type or string*) – the desired type

Returns self

Return type Module

xpu(*device: Optional[Union[int, torch.device]] = None*) → torch.nn.modules.module.T
Moves all model parameters and buffers to the XPU.

This also makes associated parameters and buffers different objects. So it should be called before constructing optimizer if the module will live on XPU while being optimized.

Note: This method modifies the module in-place.

Parameters **device** (*int, optional*) – if specified, all parameters will be copied to that device

Returns self

Return type Module

zero_grad(*set_to_none: bool = False*) → None
Sets gradients of all model parameters to zero. See similar function under `torch.optim.Optimizer` for more context.

Parameters **set_to_none** (*bool*) – instead of setting to zero, set the grads to None. See `torch.optim.Optimizer.zero_grad()` for details.

12.1.3 Survival analysis models

synthcity.plugins.core.models.survival_analysis.surv_aft module

class WeibullAFTSurvivalAnalysis(*device: Any = device(type='cpu')*, ***kwargs: Any*)
Bases: `synthcity.plugins.core.models.survival_analysis._base.SurvivalAnalysisPlugin`

fit(*X: pandas.core.frame.DataFrame*, *T: pandas.core.series.Series*, *Y: pandas.core.series.Series*) → synthcity.plugins.core.models.survival_analysis._base.SurvivalAnalysisPlugin
Training logic

static hyperparameter_space(***kwargs: Any*) → List[*synthcity.plugins.core.distribution.Distribution*]
Returns the hyperparameter space for the derived plugin.

static load(*buff: bytes*) → Any

static load_dict(*representation: dict*) → Any

static name() → str
The name of the plugin.

predict(*X: pandas.core.frame.DataFrame*, *time_horizons: List*) → pandas.core.frame.DataFrame
Predict time-to-event

classmethod sample_hyperparameters(**args: Any*, ***kwargs: Any*) → Dict[str, Any]
Sample value from the hyperparameter space for the current plugin.

save() → bytes

save_dict() → dict

save_to_file(*path: pathlib.Path*) → bytes

static version() → str
API version

synthcity.plugins.core.models.survival_analysis.surv_coxph module

class CoxPHSurvivalAnalysis(*device: Any = device(type='cpu')*, *alpha: float = 0.05*, *fit_options: dict = {'step_size': 0.1}*, ***kwargs: Any*)
Bases: `synthcity.plugins.core.models.survival_analysis._base.SurvivalAnalysisPlugin`

fit(*X: pandas.core.frame.DataFrame*, *T: pandas.core.series.Series*, *Y: pandas.core.series.Series*) → synthcity.plugins.core.models.survival_analysis._base.SurvivalAnalysisPlugin
Training logic

static hyperparameter_space(***kwargs: Any*) → List[*synthcity.plugins.core.distribution.Distribution*]
Returns the hyperparameter space for the derived plugin.

static load(*buff: bytes*) → Any

static load_dict(*representation: dict*) → Any

static name() → str
The name of the plugin.

predict(*X: pandas.core.frame.DataFrame*, *time_horizons: List*) → pandas.core.frame.DataFrame
Predict risk estimation

classmethod sample_hyperparameters(**args: Any*, ***kwargs: Any*) → Dict[str, Any]
Sample value from the hyperparameter space for the current plugin.

save() → bytes

save_dict() → dict

save_to_file(*path: pathlib.Path*) → bytes

static version() → str
API version

synthcity.plugins.core.models.survival_analysis.surv_deephit module

class DeephitSurvivalAnalysis(*num_durations: int = 500*, *batch_size: int = 100*, *epochs: int = 2000*, *lr: float = 0.01*, *dim_hidden: int = 300*, *alpha: float = 0.28*, *sigma: float = 0.38*, *dropout: float = 0.2*, *patience: int = 20*, *batch_norm: bool = False*, *random_state: int = 0*, *device: Any = device(type='cpu')*, ***kwargs: Any*)
Bases: `synthcity.plugins.core.models.survival_analysis._base.SurvivalAnalysisPlugin`

fit(*X: pandas.core.frame.DataFrame*, *T: pandas.core.series.Series*, *E: pandas.core.series.Series*) → synthcity.plugins.core.models.survival_analysis._base.SurvivalAnalysisPlugin
Training logic

static hyperparameter_space(**args: Any*, ***kwargs: Any*) → List[*synthcity.plugins.core.distribution.Distribution*]
Returns the hyperparameter space for the derived plugin.

static load(*buff: bytes*) → Any

static load_dict(*representation: dict*) → Any

static name() → str
The name of the plugin.

predict(*X: pandas.core.frame.DataFrame*, *time_horizons: List*) → pandas.core.frame.DataFrame
Predict risk

classmethod sample_hyperparameters(**args: Any*, ***kwargs: Any*) → Dict[str, Any]
Sample value from the hyperparameter space for the current plugin.

save() → bytes

save_dict() → dict

save_to_file(*path: pathlib.Path*) → bytes

static version() → str
API version

synthcity.plugins.core.models.survival_analysis.surv_xgb module

class XGBSurvivalAnalysis(*n_estimators: int = 100*, *colsample_bynode: float = 0.5*, *max_depth: int = 5*, *subsample: float = 0.5*, *learning_rate: float = 0.05*, *min_child_weight: int = 50*, *tree_method: str = 'hist'*, *booster: int = 0*, *random_state: int = 0*, *objective: str = 'aft'*, *strategy: str = 'debiased_bce'*, *bce_n_iter: int = 1000*, *time_points: int = 100*, *device: Any = device(type='cpu')*, ***kwargs: Any*)

Bases: `synthcity.plugins.core.models.survival_analysis._base.SurvivalAnalysisPlugin`

booster = ['gbtree', 'gblinear', 'dart']

explain(*X: pandas.core.frame.DataFrame*) → numpy.ndarray

fit(*X: pandas.core.frame.DataFrame*, *T: pandas.core.series.Series*, *Y: pandas.core.series.Series*) → synthcity.plugins.core.models.survival_analysis._base.SurvivalAnalysisPlugin
Training logic

static hyperparameter_space(***kwargs: Any*) → List[*synthcity.plugins.core.distribution.Distribution*]
Returns the hyperparameter space for the derived plugin.

static load(*buff: bytes*) → Any

static load_dict(*representation: dict*) → Any

static name() → str
The name of the plugin.

predict(*X: pandas.core.frame.DataFrame*, *time_horizons: List*) → pandas.core.frame.DataFrame
Predict risk

classmethod sample_hyperparameters(**args: Any*, ***kwargs: Any*) → Dict[str, Any]
Sample value from the hyperparameter space for the current plugin.

save() → bytes

save_dict() → dict

save_to_file(*path: pathlib.Path*) → bytes

static version() → str
API version

12.1.4 Time-series survival models

synthcity.plugins.core.models.time_series_survival.ts_surv_coxph module

class CoxTimeSeriesSurvival(*alpha: float = 0.05*, *penalizer: float = 0.1*, *device: Any = device(type='cpu')*, *emb_n_iter: int = 1000*, *emb_batch_size: int = 100*, *emb_lr: float = 0.001*, *emb_n_layers_hidden: int = 1*, *emb_n_units_hidden: int = 40*, *emb_split: int = 100*, *emb_rnn_type: str = 'GRU'*, *emb_output_type: str = 'MLP'*, *emb_alpha: float = 0.34*, *emb_beta: float = 0.27*, *emb_sigma: float = 0.21*, *emb_dropout: float = 0.06*, *emb_patience: int = 20*, *n_iter: Optional[int] = None*, *random_state: int = 0*)

Bases: `synthcity.plugins.core.models.time_series_survival._base.TimeSeriesSurvivalPlugin`

fit(*static: Optional[numpy.ndarray]*, *temporal: numpy.ndarray*, *observation_times: numpy.ndarray*, *T: numpy.ndarray*, *E: numpy.ndarray*) → synthcity.plugins.core.models.time_series_survival._base.TimeSeriesSurvivalPlugin
Training logic

static hyperparameter_space(**args: Any*, ***kwargs: Any*) → List[*synthcity.plugins.core.distribution.Distribution*]
Returns the hyperparameter space for the derived plugin.

static load(*buff: bytes*) → Any

static load_dict(*representation: dict*) → Any

static name() → str
The name of the plugin.

predict(*static: Optional[numpy.ndarray]*, *temporal: numpy.ndarray*, *observation_times: numpy.ndarray*, *time_horizons: List*) → numpy.ndarray
Predict risk

classmethod sample_hyperparameters(**args: Any*, ***kwargs: Any*) → Dict[str, Any]
Sample value from the hyperparameter space for the current plugin.

save() → bytes

save_dict() → dict

save_to_file(*path: pathlib.Path*) → bytes

static version() → str
API version

synthcity.plugins.core.models.time_series_survival.ts_surv_dynamic_deephit module

class DynamicDeepHitLayers(*input_dim: int*, *seq_len: int*, *output_dim: int*, *layers_rnn: int*, *hidden_rnn: int*, *rnn_type: str = 'LSTM'*, *dropout: float = 0.1*, *risks: int = 1*, *output_type: str = 'MLP'*, *device: Any = device(type='cpu')*)
Bases: `torch.nn.modules.module.Module`

T_destination
alias of TypeVar('T_destination', bound=`Dict[str, Any]`)

add_module(*name: str*, *module: Optional[torch.nn.modules.module.Module]*) → None
Adds a child module to the current module.

The module can be accessed as an attribute using the given name.

Parameters

- **name** (*str*) – name of the child module. The child module can be accessed from this module using the given name
- **module** (*Module*) – child module to be added to the module.

apply(*fn: Callable[[torch.nn.modules.module.Module], None]*) → torch.nn.modules.module.T
Applies `fn` recursively to every submodule (as returned by `.children()`) as well as self. Typical use includes initializing the parameters of a model (see also nn-init-doc).

Parameters **fn** (`Module` -> None) – function to be applied to each submodule

Returns self

Return type Module

Example:

```
>>> @torch.no_grad()
>>> def init_weights(m):
>>>     print(m)
>>>     if type(m) == nn.Linear:
>>>         m.weight.fill_(1.0)
>>>         print(m.weight)
>>> net = nn.Sequential(nn.Linear(2, 2), nn.Linear(2, 2))
>>> net.apply(init_weights)
Linear(in_features=2, out_features=2, bias=True)
Parameter containing:
tensor([[1., 1.],
        [1., 1.]], requires_grad=True)
Linear(in_features=2, out_features=2, bias=True)
Parameter containing:
tensor([[1., 1.],
        [1., 1.]], requires_grad=True)
Sequential(
  (0): Linear(in_features=2, out_features=2, bias=True)
  (1): Linear(in_features=2, out_features=2, bias=True)
)
```

bfloat16() → torch.nn.modules.module.T
Casts all floating point parameters and buffers to `bfloat16` datatype.

Note: This method modifies the module in-place.

Returns self

Return type Module

buffers(*recurse: bool = True*) → Iterator[torch.Tensor]
Returns an iterator over module buffers.

Parameters **recurse** (*bool*) – if True, then yields buffers of this module and all submodules. Otherwise, yields only buffers that are direct members of this module.

Yields *torch.Tensor* – module buffer

Example:

```
>>> # xdoctest: +SKIP("undefined vars")
>>> for buf in model.buffers():
>>>     print(type(buf), buf.size())
<class 'torch.Tensor'> (20L,)
<class 'torch.Tensor'> (20L, 1L, 5L, 5L)
```

children() → Iterator[torch.nn.modules.module.Module]
Returns an iterator over immediate children modules.

Yields *Module* – a child module

cpu() → torch.nn.modules.module.T
Moves all model parameters and buffers to the CPU.

Note: This method modifies the module in-place.

Returns self

Return type Module

`cuda`(*device: Optional[Union[int, torch.device]] = None*) → torch.nn.modules.module.T
Moves all model parameters and buffers to the GPU.

This also makes associated parameters and buffers different objects. So it should be called before constructing optimizer if the module will live on GPU while being optimized.

Note: This method modifies the module in-place.

Parameters `device` (*int, optional*) – if specified, all parameters will be copied to that device

Returns self

Return type Module

`double`() → torch.nn.modules.module.T
Casts all floating point parameters and buffers to `double` datatype.

Note: This method modifies the module in-place.

Returns self

Return type Module

`dump_patches: bool = False`

`eval`() → torch.nn.modules.module.T
Sets the module in evaluation mode.

This has any effect only on certain modules. See documentations of particular modules for details of their behaviors in training/evaluation mode, if they are affected, e.g. `Dropout`, `BatchNorm`, etc.

This is equivalent with `self.train(False)`.

See locally-disable-grad-doc for a comparison between *.eval()* and several similar mechanisms that may be confused with it.

Returns self

Return type Module

`extra_repr`() → str
Set the extra representation of the module

To print customized extra information, you should re-implement this method in your own modules. Both single-line and multi-line strings are acceptable.

`float`() → torch.nn.modules.module.T
Casts all floating point parameters and buffers to `float` datatype.

Note: This method modifies the module in-place.

Returns self

Return type Module

forward(*x: torch.Tensor*) → torch.Tensor
The forward function that is called when data is passed through DynamicDeepHit.

forward_attention(*x: torch.Tensor*, *inputmask: torch.Tensor*, *hidden: torch.Tensor*) → torch.Tensor

forward_emb(*x: torch.Tensor*) → torch.Tensor
The forward function that is called when data is passed through DynamicDeepHit.

get_buffer(*target: str*) → torch.Tensor
Returns the buffer given by `target` if it exists, otherwise throws an error.

See the docstring for `get_submodule` for a more detailed explanation of this method's functionality as well as how to correctly specify `target`.

Parameters **target** – The fully-qualified string name of the buffer to look for. (See `get_submodule` for how to specify a fully-qualified string.)

Returns The buffer referenced by `target`

Return type torch.Tensor

Raises **AttributeError** – If the target string references an invalid path or resolves to something that is not a buffer

get_extra_state() → Any
Returns any extra state to include in the module's state_dict. Implement this and a corresponding *set_extra_state()* for your module if you need to store extra state. This function is called when building the module's *state_dict()*.

Note that extra state should be pickleable to ensure working serialization of the state_dict. We only provide provide backwards compatibility guarantees for serializing Tensors; other objects may break backwards compatibility if their serialized pickled form changes.

Returns Any extra state to store in the module's state_dict

Return type object

get_parameter(*target: str*) → torch.nn.parameter.Parameter
Returns the parameter given by `target` if it exists, otherwise throws an error.

See the docstring for `get_submodule` for a more detailed explanation of this method's functionality as well as how to correctly specify `target`.

Parameters **target** – The fully-qualified string name of the Parameter to look for. (See `get_submodule` for how to specify a fully-qualified string.)

Returns The Parameter referenced by `target`

Return type torch.nn.Parameter

Raises **AttributeError** – If the target string references an invalid path or resolves to something that is not an `nn.Parameter`

get_submodule(*target: str*) → torch.nn.modules.module.Module
Returns the submodule given by `target` if it exists, otherwise throws an error.

For example, let's say you have an `nn.Module` `A` that looks like this:

```
A(
    (net_b): Module(
        (net_c): Module(
            (conv): Conv2d(16, 33, kernel_size=(3, 3), stride=(2, 2))
        )
        (linear): Linear(in_features=100, out_features=200, bias=True)
    )
)
```

(The diagram shows an `nn.Module` `A`. `A` has a nested submodule `net_b`, which itself has two submodules `net_c` and `linear`. `net_c` then has a submodule `conv`.)

To check whether or not we have the `linear` submodule, we would call `get_submodule("net_b.linear")`. To check whether we have the `conv` submodule, we would call `get_submodule("net_b.net_c.conv")`.

The runtime of `get_submodule` is bounded by the degree of module nesting in `target`. A query against `named_modules` achieves the same result, but it is O(N) in the number of transitive modules. So, for a simple check to see if some submodule exists, `get_submodule` should always be used.

Parameters `target` – The fully-qualified string name of the submodule to look for. (See above example for how to specify a fully-qualified string.)

Returns The submodule referenced by `target`

Return type torch.nn.Module

Raises `AttributeError` – If the target string references an invalid path or resolves to something that is not an `nn.Module`

`half`() → torch.nn.modules.module.T

Casts all floating point parameters and buffers to `half` datatype.

Note: This method modifies the module in-place.

Returns self

Return type Module

`ipu`(*device: Optional[Union[int, torch.device]] = None*) → torch.nn.modules.module.T

Moves all model parameters and buffers to the IPU.

This also makes associated parameters and buffers different objects. So it should be called before constructing optimizer if the module will live on IPU while being optimized.

Note: This method modifies the module in-place.

Parameters `device` (*int, optional*) – if specified, all parameters will be copied to that device

Returns self

Return type Module

load_state_dict(*state_dict: Mapping[str, Any]*, *strict: bool = True*)

Copies parameters and buffers from *state_dict* into this module and its descendants. If `strict` is `True`, then the keys of *state_dict* must exactly match the keys returned by this module's `state_dict()` function.

Parameters

- **state_dict** (*dict*) – a dict containing parameters and persistent buffers.
- **strict** (*bool, optional*) – whether to strictly enforce that the keys in *state_dict* match the keys returned by this module's `state_dict()` function. Default: `True`

Returns

- **missing_keys** is a list of str containing the missing keys
- **unexpected_keys** is a list of str containing the unexpected keys

Return type `NamedTuple` with `missing_keys` and `unexpected_keys` fields

Note: If a parameter or buffer is registered as `None` and its corresponding key exists in *state_dict*, *load_state_dict()* will raise a `RuntimeError`.

---

modules() → Iterator[torch.nn.modules.module.Module]

Returns an iterator over all modules in the network.

Yields *Module* – a module in the network

Note: Duplicate modules are returned only once. In the following example, `l` will be returned only once.

Example:

```
>>> l = nn.Linear(2, 2)
>>> net = nn.Sequential(l, l)
>>> for idx, m in enumerate(net.modules()):
...     print(idx, '->', m)

0 -> Sequential(
  (0): Linear(in_features=2, out_features=2, bias=True)
  (1): Linear(in_features=2, out_features=2, bias=True)
)
1 -> Linear(in_features=2, out_features=2, bias=True)
```

named_buffers(*prefix: str = ''*, *recurse: bool = True*) → Iterator[Tuple[str, torch.Tensor]]

Returns an iterator over module buffers, yielding both the name of the buffer as well as the buffer itself.

Parameters

- **prefix** (*str*) – prefix to prepend to all buffer names.
- **recurse** (*bool*) – if True, then yields buffers of this module and all submodules. Otherwise, yields only buffers that are direct members of this module.

Yields *(str, torch.Tensor)* – Tuple containing the name and buffer

Example:

```
>>> # xdoctest: +SKIP("undefined vars")
>>> for name, buf in self.named_buffers():
>>>    if name in ['running_var']:
>>>        print(buf.size())
```

named_children() → Iterator[Tuple[str, torch.nn.modules.module.Module]]

Returns an iterator over immediate children modules, yielding both the name of the module as well as the module itself.

Yields *(str, Module)* – Tuple containing a name and child module

Example:

```
>>> # xdoctest: +SKIP("undefined vars")
>>> for name, module in model.named_children():
>>>     if name in ['conv4', 'conv5']:
>>>         print(module)
```

named_modules(*memo: Optional[Set[torch.nn.modules.module.Module]] = None*, *prefix: str = ''*, *remove_duplicate: bool = True*)

Returns an iterator over all modules in the network, yielding both the name of the module as well as the module itself.

Parameters

- **memo** – a memo to store the set of modules already added to the result
- **prefix** – a prefix that will be added to the name of the module
- **remove_duplicate** – whether to remove the duplicated module instances in the result or not

Yields *(str, Module)* – Tuple of name and module

Note: Duplicate modules are returned only once. In the following example, `l` will be returned only once.

Example:

```
>>> l = nn.Linear(2, 2)
>>> net = nn.Sequential(l, l)
>>> for idx, m in enumerate(net.named_modules()):
...     print(idx, '->', m)

0 -> ('', Sequential(
  (0): Linear(in_features=2, out_features=2, bias=True)
  (1): Linear(in_features=2, out_features=2, bias=True)
))
1 -> ('0', Linear(in_features=2, out_features=2, bias=True))
```

named_parameters(*prefix: str = ''*, *recurse: bool = True*) → Iterator[Tuple[str, torch.nn.parameter.Parameter]]

Returns an iterator over module parameters, yielding both the name of the parameter as well as the parameter itself.

Parameters

- **prefix** (*str*) – prefix to prepend to all parameter names.

- **recurse** (*bool*) – if True, then yields parameters of this module and all submodules. Otherwise, yields only parameters that are direct members of this module.

Yields *(str, Parameter)* – Tuple containing the name and parameter

Example:

```
>>> # xdoctest: +SKIP("undefined vars")
>>> for name, param in self.named_parameters():
>>>    if name in ['bias']:
>>>        print(param.size())
```

parameters(*recurse: bool = True*) → Iterator[torch.nn.parameter.Parameter]

Returns an iterator over module parameters.

This is typically passed to an optimizer.

Parameters **recurse** (*bool*) – if True, then yields parameters of this module and all submodules. Otherwise, yields only parameters that are direct members of this module.

Yields *Parameter* – module parameter

Example:

```
>>> # xdoctest: +SKIP("undefined vars")
>>> for param in model.parameters():
>>>     print(type(param), param.size())
<class 'torch.Tensor'> (20L,)
<class 'torch.Tensor'> (20L, 1L, 5L, 5L)
```

register_backward_hook(*hook: Callable[[torch.nn.modules.module.Module, Union[Tuple[torch.Tensor, ...], torch.Tensor], Union[Tuple[torch.Tensor, ...], torch.Tensor]], Union[None, torch.Tensor]]*) → torch.utils.hooks.RemovableHandle

Registers a backward hook on the module.

This function is deprecated in favor of `register_full_backward_hook()` and the behavior of this function will change in future versions.

Returns a handle that can be used to remove the added hook by calling `handle.remove()`

Return type `torch.utils.hooks.RemovableHandle`

register_buffer(*name: str*, *tensor: Optional[torch.Tensor]*, *persistent: bool = True*) → None

Adds a buffer to the module.

This is typically used to register a buffer that should not to be considered a model parameter. For example, BatchNorm's `running_mean` is not a parameter, but is part of the module's state. Buffers, by default, are persistent and will be saved alongside parameters. This behavior can be changed by setting `persistent` to `False`. The only difference between a persistent buffer and a non-persistent buffer is that the latter will not be a part of this module's *state_dict*.

Buffers can be accessed as attributes using given names.

Parameters

- **name** (*str*) – name of the buffer. The buffer can be accessed from this module using the given name
- **tensor** (*Tensor or None*) – buffer to be registered. If `None`, then operations that run on buffers, such as *cuda*, are ignored. If `None`, the buffer is **not** included in the module's *state_dict*.

- **persistent** (*bool*) – whether the buffer is part of this module's *state_dict*.

Example:

```
>>> # xdoctest: +SKIP("undefined vars")
>>> self.register_buffer('running_mean', torch.zeros(num_features))
```

register_forward_hook(*hook: Callable[[...], None]*) → torch.utils.hooks.RemovableHandle

Registers a forward hook on the module.

The hook will be called every time after *forward()* has computed an output. It should have the following signature:

```
hook(module, input, output) -> None or modified output
```

The input contains only the positional arguments given to the module. Keyword arguments won't be passed to the hooks and only to the `forward`. The hook can modify the output. It can modify the input inplace but it will not have effect on forward since this is called after *forward()* is called.

Returns a handle that can be used to remove the added hook by calling `handle.remove()`

Return type `torch.utils.hooks.RemovableHandle`

register_forward_pre_hook(*hook: Callable[[...], None]*) → torch.utils.hooks.RemovableHandle

Registers a forward pre-hook on the module.

The hook will be called every time before *forward()* is invoked. It should have the following signature:

```
hook(module, input) -> None or modified input
```

The input contains only the positional arguments given to the module. Keyword arguments won't be passed to the hooks and only to the `forward`. The hook can modify the input. User can either return a tuple or a single modified value in the hook. We will wrap the value into a tuple if a single value is returned(unless that value is already a tuple).

Returns a handle that can be used to remove the added hook by calling `handle.remove()`

Return type `torch.utils.hooks.RemovableHandle`

register_full_backward_hook(*hook: Callable[[torch.nn.modules.module.Module, Union[Tuple[torch.Tensor, ...], torch.Tensor], Union[Tuple[torch.Tensor, ...], torch.Tensor]], Union[None, torch.Tensor]]*) → torch.utils.hooks.RemovableHandle

Registers a backward hook on the module.

The hook will be called every time the gradients with respect to a module are computed, i.e. the hook will execute if and only if the gradients with respect to module outputs are computed. The hook should have the following signature:

```
hook(module, grad_input, grad_output) -> tuple(Tensor) or None
```

The `grad_input` and `grad_output` are tuples that contain the gradients with respect to the inputs and outputs respectively. The hook should not modify its arguments, but it can optionally return a new gradient with respect to the input that will be used in place of `grad_input` in subsequent computations. `grad_input` will only correspond to the inputs given as positional arguments and all kwarg arguments are ignored. Entries in `grad_input` and `grad_output` will be `None` for all non-Tensor arguments.

For technical reasons, when this hook is applied to a Module, its forward function will receive a view of each Tensor passed to the Module. Similarly the caller will receive a view of each Tensor returned by the Module's forward function.

> **Warning:** Modifying inputs or outputs inplace is not allowed when using backward hooks and will raise an error.

Returns a handle that can be used to remove the added hook by calling `handle.remove()`

Return type `torch.utils.hooks.RemovableHandle`

register_load_state_dict_post_hook(*hook*)

Registers a post hook to be run after module's `load_state_dict` is called.

It should have the following signature:: hook(module, incompatible_keys) -> None

The `module` argument is the current module that this hook is registered on, and the `incompatible_keys` argument is a `NamedTuple` consisting of attributes `missing_keys` and `unexpected_keys`. `missing_keys` is a `list` of `str` containing the missing keys and `unexpected_keys` is a `list` of `str` containing the unexpected keys.

The given incompatible_keys can be modified inplace if needed.

Note that the checks performed when calling *load_state_dict()* with `strict=True` are affected by modifications the hook makes to `missing_keys` or `unexpected_keys`, as expected. Additions to either set of keys will result in an error being thrown when `strict=True`, and clearning out both missing and unexpected keys will avoid an error.

Returns a handle that can be used to remove the added hook by calling `handle.remove()`

Return type `torch.utils.hooks.RemovableHandle`

register_module(*name: str*, *module: Optional[torch.nn.modules.module.Module]*) → None

Alias for *add_module()*.

register_parameter(*name: str*, *param: Optional[torch.nn.parameter.Parameter]*) → None

Adds a parameter to the module.

The parameter can be accessed as an attribute using given name.

Parameters

- **name** (*str*) – name of the parameter. The parameter can be accessed from this module using the given name
- **param** (*Parameter or None*) – parameter to be added to the module. If `None`, then operations that run on parameters, such as *cuda*, are ignored. If `None`, the parameter is **not** included in the module's *state_dict*.

requires_grad_(*requires_grad: bool = True*) → torch.nn.modules.module.T

Change if autograd should record operations on parameters in this module.

This method sets the parameters' `requires_grad` attributes in-place.

This method is helpful for freezing part of the module for finetuning or training parts of a model individually (e.g., GAN training).

See locally-disable-grad-doc for a comparison between *.requires_grad_()* and several similar mechanisms that may be confused with it.

Parameters **requires_grad** (*bool*) – whether autograd should record operations on parameters in this module. Default: `True`.

Returns self

Return type Module

set_extra_state(*state: Any*)
This function is called from *load_state_dict()* to handle any extra state found within the *state_dict*. Implement this function and a corresponding *get_extra_state()* for your module if you need to store extra state within its *state_dict*.

Parameters **state** (*dict*) – Extra state from the *state_dict*

share_memory() → torch.nn.modules.module.T
See `torch.Tensor.share_memory_()`

state_dict(**args*, *destination=None*, *prefix=''*, *keep_vars=False*)
Returns a dictionary containing references to the whole state of the module.

Both parameters and persistent buffers (e.g. running averages) are included. Keys are corresponding parameter and buffer names. Parameters and buffers set to `None` are not included.

Note: The returned object is a shallow copy. It contains references to the module's parameters and buffers.

Warning: Currently `state_dict()` also accepts positional arguments for `destination`, `prefix` and `keep_vars` in order. However, this is being deprecated and keyword arguments will be enforced in future releases.

Warning: Please avoid the use of argument `destination` as it is not designed for end-users.

Parameters

- **destination** (*dict, optional*) – If provided, the state of module will be updated into the dict and the same object is returned. Otherwise, an `OrderedDict` will be created and returned. Default: `None`.
- **prefix** (*str, optional*) – a prefix added to parameter and buffer names to compose the keys in state_dict. Default: `''`.
- **keep_vars** (*bool, optional*) – by default the `Tensor` s returned in the state dict are detached from autograd. If it's set to `True`, detaching will not be performed. Default: `False`.

Returns a dictionary containing a whole state of the module

Return type dict

Example:

```
>>> # xdoctest: +SKIP("undefined vars")
>>> module.state_dict().keys()
['bias', 'weight']
```

to(**args*, ***kwargs*)
Moves and/or casts the parameters and buffers.

This can be called as

to(*device=None*, *dtype=None*, *non_blocking=False*)

to(*dtype*, *non_blocking=False*)

to(*tensor*, *non_blocking=False*)

to(*memory_format=torch.channels_last*)

Its signature is similar to `torch.Tensor.to()`, but only accepts floating point or complex `dtype`s. In addition, this method will only cast the floating point or complex parameters and buffers to `dtype` (if given). The integral parameters and buffers will be moved `device`, if that is given, but with dtypes unchanged. When `non_blocking` is set, it tries to convert/move asynchronously with respect to the host if possible, e.g., moving CPU Tensors with pinned memory to CUDA devices.

See below for examples.

Note: This method modifies the module in-place.

Parameters

- **device** (`torch.device`) – the desired device of the parameters and buffers in this module
- **dtype** (`torch.dtype`) – the desired floating point or complex dtype of the parameters and buffers in this module
- **tensor** (*torch.Tensor*) – Tensor whose dtype and device are the desired dtype and device for all parameters and buffers in this module
- **memory_format** (`torch.memory_format`) – the desired memory format for 4D parameters and buffers in this module (keyword only argument)

Returns self

Return type Module

Examples:

```
>>> # xdoctest: +IGNORE_WANT("non-deterministic")
>>> linear = nn.Linear(2, 2)
>>> linear.weight
Parameter containing:
tensor([[ 0.1913, -0.3420],
        [-0.5113, -0.2325]])
>>> linear.to(torch.double)
Linear(in_features=2, out_features=2, bias=True)
>>> linear.weight
Parameter containing:
tensor([[ 0.1913, -0.3420],
        [-0.5113, -0.2325]], dtype=torch.float64)
>>> # xdoctest: +REQUIRES(env:TORCH_DOCTEST_CUDA1)
>>> gpu1 = torch.device("cuda:1")
>>> linear.to(gpu1, dtype=torch.half, non_blocking=True)
Linear(in_features=2, out_features=2, bias=True)
>>> linear.weight
Parameter containing:
tensor([[ 0.1914, -0.3420],
        [-0.5112, -0.2324]], dtype=torch.float16, device='cuda:1')
>>> cpu = torch.device("cpu")
>>> linear.to(cpu)
```

(continues on next page)

(continued from previous page)

```
Linear(in_features=2, out_features=2, bias=True)
>>> linear.weight
Parameter containing:
tensor([[ 0.1914, -0.3420],
        [-0.5112, -0.2324]], dtype=torch.float16)

>>> linear = nn.Linear(2, 2, bias=None).to(torch.cdouble)
>>> linear.weight
Parameter containing:
tensor([[ 0.3741+0.j,  0.2382+0.j],
        [ 0.5593+0.j, -0.4443+0.j]], dtype=torch.complex128)
>>> linear(torch.ones(3, 2, dtype=torch.cdouble))
tensor([[0.6122+0.j, 0.1150+0.j],
        [0.6122+0.j, 0.1150+0.j],
        [0.6122+0.j, 0.1150+0.j]], dtype=torch.complex128)
```

to_empty(*, *device: Union[str, torch.device]*) → torch.nn.modules.module.T
Moves the parameters and buffers to the specified device without copying storage.

Parameters **device** (`torch.device`) – The desired device of the parameters and buffers in this module.

Returns self

Return type Module

train(*mode: bool = True*) → torch.nn.modules.module.T
Sets the module in training mode.

This has any effect only on certain modules. See documentations of particular modules for details of their behaviors in training/evaluation mode, if they are affected, e.g. `Dropout`, `BatchNorm`, etc.

Parameters **mode** (*bool*) – whether to set training mode (`True`) or evaluation mode (`False`). Default: `True`.

Returns self

Return type Module

training: bool

type(*dst_type: Union[torch.dtype, str]*) → torch.nn.modules.module.T
Casts all parameters and buffers to `dst_type`.

Note: This method modifies the module in-place.

Parameters **dst_type** (*type or string*) – the desired type

Returns self

Return type Module

xpu(*device: Optional[Union[int, torch.device]] = None*) → torch.nn.modules.module.T
Moves all model parameters and buffers to the XPU.

This also makes associated parameters and buffers different objects. So it should be called before constructing optimizer if the module will live on XPU while being optimized.

Note: This method modifies the module in-place.

Parameters **device** (*int, optional*) – if specified, all parameters will be copied to that device

Returns self

Return type Module

zero_grad(*set_to_none: bool = False*) → None
Sets gradients of all model parameters to zero. See similar function under `torch.optim.Optimizer` for more context.

Parameters **set_to_none** (*bool*) – instead of setting to zero, set the grads to None. See `torch.optim.Optimizer.zero_grad()` for details.

class DynamicDeepHitModel(*split: int = 100*, *layers_rnn: int = 2*, *hidden_rnn: int = 100*, *rnn_type: str = 'LSTM'*, *dropout: float = 0.1*, *alpha: float = 0.1*, *beta: float = 0.1*, *sigma: float = 0.1*, *patience: int = 20*, *lr: float = 0.001*, *batch_size: int = 100*, *n_iter: int = 1000*, *device: Any = device(type='cpu')*, *val_size: float = 0.1*, *random_state: int = 0*, *clipping_value: int = 1*, *output_type: str = 'MLP'*)
Bases: `object`

This implementation considers that the last event happen at the same time for each patient The CIF is therefore simplified

discretize(*t: numpy.ndarray*, *split: int*, *split_time: Optional[int] = None*) → Tuple
Discretize the survival horizon

Parameters

- **t** (*List of Array*) – Time of events
- **split** (*int*) – Number of bins
- **split_time** (*List, optional*) – List of bins (must be same length than split). Defaults to None.

Returns Disretized events time

Return type List of Array

fit(*x: numpy.ndarray*, *t: numpy.ndarray*, *e: numpy.ndarray*) → Any

longitudinal_loss(*longitudinal_prediction: torch.Tensor*, *x: torch.Tensor*) → torch.Tensor
Penalize error in the longitudinal predictions This function is used to compute the error made by the RNN

NB: In the paper, they seem to use different losses for continuous and categorical But this was not reflected in the code associated (therefore we compute MSE for all)

NB: Original paper mentions possibility of different alphas for each risk But take same for all (for ranking loss)

negative_log_likelihood(*outcomes: torch.Tensor*, *cif: torch.Tensor*, *t: torch.Tensor*, *e: torch.Tensor*) → torch.Tensor
Compute the log likelihood loss This function is used to compute the survival loss

predict_emb(*x: numpy.ndarray*) → numpy.ndarray

predict_risk(*x: numpy.ndarray*, *t: numpy.ndarray*, ***args: Any*) → numpy.ndarray

predict_survival(*x: numpy.ndarray*, *t: numpy.ndarray*, *risk: int = 1*, *all_step: bool = False*, *bs: int = 100*) → numpy.ndarray

ranking_loss(*cif: torch.Tensor*, *t: torch.Tensor*, *e: torch.Tensor*) → torch.Tensor
Penalize wrong ordering of probability Equivalent to a C Index This function is used to penalize wrong ordering in the survival prediction

total_loss(*x: torch.Tensor*, *t: torch.Tensor*, *e: torch.Tensor*) → torch.Tensor

class DynamicDeephitTimeSeriesSurvival(*n_iter: int = 1000*, *batch_size: int = 100*, *lr: float = 0.001*, *n_layers_hidden: int = 1*, *n_units_hidden: int = 40*, *split: int = 100*, *rnn_type: str = 'GRU'*, *alpha: float = 0.34*, *beta: float = 0.27*, *sigma: float = 0.21*, *random_state: int = 0*, *dropout: float = 0.06*, *device: Any = device(type='cpu')*, *patience: int = 20*, *output_type: str = 'MLP'*, ***kwargs: Any*)
Bases: synthcity.plugins.core.models.time_series_survival._base.TimeSeriesSurvivalPlugin

fit(*static: Optional[numpy.ndarray]*, *temporal: Union[numpy.ndarray, List]*, *observation_times: Union[numpy.ndarray, List]*, *T: Union[numpy.ndarray, List]*, *E: Union[numpy.ndarray, List]*) → synthcity.plugins.core.models.time_series_survival._base.TimeSeriesSurvivalPlugin
Training logic

static hyperparameter_space(**args: Any*, *prefix: str = ''*, ***kwargs: Any*) → List[*synthcity.plugins.core.distribution.Distribution*]
Returns the hyperparameter space for the derived plugin.

static load(*buff: bytes*) → Any

static load_dict(*representation: dict*) → Any

static name() → str
The name of the plugin.

predict(*static: Optional[numpy.ndarray]*, *temporal: Union[numpy.ndarray, List]*, *observation_times: Union[numpy.ndarray, List]*, *time_horizons: List*, *batch_size: Optional[int] = 100*) → numpy.ndarray
Predict risk

predict_emb(*static: Optional[numpy.ndarray]*, *temporal: Union[numpy.ndarray, List]*, *observation_times: Union[numpy.ndarray, List]*) → numpy.ndarray
Predict embeddings

classmethod sample_hyperparameters(**args: Any*, ***kwargs: Any*) → Dict[str, Any]
Sample value from the hyperparameter space for the current plugin.

save() → bytes

save_dict() → dict

save_to_file(*path: pathlib.Path*) → bytes

static version() → str
API version

synthcity.plugins.core.models.time_series_survival.ts_surv_xgb module

class XGBTimeSeriesSurvival(*n_estimators: int = 100*, *colsample_bynode: float = 0.5*, *max_depth: int = 5*, *subsample: float = 0.5*, *learning_rate: float = 0.05*, *min_child_weight: int = 50*, *tree_method: str = 'hist'*, *booster: int = 0*, *random_state: int = 0*, *objective: str = 'aft'*, *strategy: str = 'km'*, *bce_n_iter: int = 1000*, *time_points: int = 100*, *device: Any = device(type='cpu')*, *emb_n_iter: int = 1000*, *emb_batch_size: int = 100*, *emb_lr: float = 0.001*, *emb_n_layers_hidden: int = 1*, *emb_n_units_hidden: int = 40*, *emb_split: int = 100*, *emb_rnn_type: str = 'GRU'*, *emb_output_type: str = 'MLP'*, *emb_alpha: float = 0.34*, *emb_beta: float = 0.27*, *emb_sigma: float = 0.21*, *emb_dropout: float = 0.06*, *emb_patience: int = 20*, *n_iter: Optional[int] = None*, ***kwargs: Any*)

Bases: synthcity.plugins.core.models.time_series_survival._base.TimeSeriesSurvivalPlugin

fit(*static: Optional[numpy.ndarray]*, *temporal: numpy.ndarray*, *observation_times: numpy.ndarray*, *T: numpy.ndarray*, *E: numpy.ndarray*) → synthcity.plugins.core.models.time_series_survival._base.TimeSeriesSurvivalPlugin

Training logic

static hyperparameter_space(**args: Any*, ***kwargs: Any*) → List[*synthcity.plugins.core.distribution.Distribution*]

Returns the hyperparameter space for the derived plugin.

static load(*buff: bytes*) → Any

static load_dict(*representation: dict*) → Any

static name() → str

The name of the plugin.

predict(*static: Optional[numpy.ndarray]*, *temporal: numpy.ndarray*, *observation_times: numpy.ndarray*, *time_horizons: List*) → numpy.ndarray

Predict risk

classmethod sample_hyperparameters(**args: Any*, ***kwargs: Any*) → Dict[str, Any]

Sample value from the hyperparameter space for the current plugin.

save() → bytes

save_dict() → dict

save_to_file(*path: pathlib.Path*) → bytes

static version() → str

API version

12.1.5 Time-to-event models

synthcity.plugins.core.models.time_to_event.tte_date module

PyTorch implementation for "Adversarial Time-to-Event Modeling" Paper: https://arxiv.org/pdf/1804.03184.pdf

class DATETimeToEvent(***kwargs: Any*)

Bases: synthcity.plugins.core.models.time_to_event._base.TimeToEventPlugin

fit(*X: pandas.core.frame.DataFrame*, *T: pandas.core.series.Series*, *Y: pandas.core.series.Series*) → synthcity.plugins.core.models.time_to_event._base.TimeToEventPlugin

Training logic

static hyperparameter_space(***kwargs: Any*) → List[*synthcity.plugins.core.distribution.Distribution*]
Returns the hyperparameter space for the derived plugin.

static load(*buff: bytes*) → Any

static load_dict(*representation: dict*) → Any

static name() → str
The name of the plugin.

predict(*X: pandas.core.frame.DataFrame*) → pandas.core.series.Series
Predict time-to-event

predict_any(*X: pandas.core.frame.DataFrame*, *E: pandas.core.series.Series*) → pandas.core.series.Series
Predict time-to-event

classmethod sample_hyperparameters(**args: Any*, ***kwargs: Any*) → Dict[str, Any]
Sample value from the hyperparameter space for the current plugin.

save() → bytes

save_dict() → dict

save_to_file(*path: pathlib.Path*) → bytes

static version() → str
API version

class TimeEventGAN(*n_features: int*, *n_units_latent: int*, *model_search_n_iter: Optional[int] = None*, *generator_n_layers_hidden: int = 2*, *generator_n_units_hidden: int = 250*, *generator_nonlin: str = 'leaky_relu'*, *generator_nonlin_out: Optional[List[Tuple[str, int]]] = None*, *generator_n_iter: int = 1000*, *generator_batch_norm: bool = False*, *generator_dropout: float = 0*, *generator_loss: Optional[Callable] = None*, *generator_lr: float = 0.0002*, *generator_weight_decay: float = 0.001*, *generator_residual: bool = True*, *generator_opt_betas: tuple = (0.9, 0.999)*, *discriminator_n_layers_hidden: int = 3*, *discriminator_n_units_hidden: int = 300*, *discriminator_nonlin: str = 'leaky_relu'*, *discriminator_n_iter: int = 1*, *discriminator_batch_norm: bool = False*, *discriminator_dropout: float = 0.1*, *discriminator_loss: Optional[Callable] = None*, *discriminator_lr: float = 0.0002*, *discriminator_weight_decay: float = 0.001*, *discriminator_opt_betas: tuple = (0.9, 0.999)*, *patience: int = 10*, *batch_size: int = 100*, *n_iter_print: int = 50*, *random_state: int = 0*, *n_iter_min: int = 100*, *clipping_value: int = 0*, *device: Any = device(type='cpu')*)
Bases: `torch.nn.modules.module.Module`

T_destination
alias of TypeVar('T_destination', bound=`Dict[str, Any]`)

add_module(*name: str*, *module: Optional[torch.nn.modules.module.Module]*) → None
Adds a child module to the current module.

The module can be accessed as an attribute using the given name.

Parameters

- **name** (*str*) – name of the child module. The child module can be accessed from this module using the given name
- **module** (*Module*) – child module to be added to the module.

apply(*fn: Callable[[torch.nn.modules.module.Module], None]*) → torch.nn.modules.module.T
Applies `fn` recursively to every submodule (as returned by `.children()`) as well as self. Typical use includes initializing the parameters of a model (see also nn-init-doc).

Parameters **fn** (Module -> None) – function to be applied to each submodule

Returns self

Return type Module

Example:

```
>>> @torch.no_grad()
>>> def init_weights(m):
>>>     print(m)
>>>     if type(m) == nn.Linear:
>>>         m.weight.fill_(1.0)
>>>         print(m.weight)
>>> net = nn.Sequential(nn.Linear(2, 2), nn.Linear(2, 2))
>>> net.apply(init_weights)
Linear(in_features=2, out_features=2, bias=True)
Parameter containing:
tensor([[1., 1.],
        [1., 1.]], requires_grad=True)
Linear(in_features=2, out_features=2, bias=True)
Parameter containing:
tensor([[1., 1.],
        [1., 1.]], requires_grad=True)
Sequential(
  (0): Linear(in_features=2, out_features=2, bias=True)
  (1): Linear(in_features=2, out_features=2, bias=True)
)
```

bfloat16() → torch.nn.modules.module.T

Casts all floating point parameters and buffers to `bfloat16` datatype.

Note: This method modifies the module in-place.

Returns self

Return type Module

buffers(*recurse: bool = True*) → Iterator[torch.Tensor]

Returns an iterator over module buffers.

Parameters **recurse** (*bool*) – if True, then yields buffers of this module and all submodules. Otherwise, yields only buffers that are direct members of this module.

Yields *torch.Tensor* – module buffer

Example:

```
>>> # xdoctest: +SKIP("undefined vars")
>>> for buf in model.buffers():
>>>     print(type(buf), buf.size())
<class 'torch.Tensor'> (20L,)
<class 'torch.Tensor'> (20L, 1L, 5L, 5L)
```

children() → Iterator[torch.nn.modules.module.Module]

Returns an iterator over immediate children modules.

Yields *Module* – a child module

cpu() → torch.nn.modules.module.T
Moves all model parameters and buffers to the CPU.

Note: This method modifies the module in-place.

Returns self

Return type Module

cuda(*device: Optional[Union[int, torch.device]] = None*) → torch.nn.modules.module.T
Moves all model parameters and buffers to the GPU.

This also makes associated parameters and buffers different objects. So it should be called before constructing optimizer if the module will live on GPU while being optimized.

Note: This method modifies the module in-place.

Parameters **device** (*int, optional*) – if specified, all parameters will be copied to that device

Returns self

Return type Module

dataloader(*X: torch.Tensor*, *T: torch.Tensor*, *E: torch.Tensor*) →
Tuple[torch.utils.data.dataloader.DataLoader, torch.utils.data.dataset.TensorDataset]

double() → torch.nn.modules.module.T
Casts all floating point parameters and buffers to `double` datatype.

Note: This method modifies the module in-place.

Returns self

Return type Module

dump_patches: bool = False

eval() → torch.nn.modules.module.T
Sets the module in evaluation mode.

This has any effect only on certain modules. See documentations of particular modules for details of their behaviors in training/evaluation mode, if they are affected, e.g. `Dropout`, `BatchNorm`, etc.

This is equivalent with `self.train(False)`.

See locally-disable-grad-doc for a comparison between *.eval()* and several similar mechanisms that may be confused with it.

Returns self

Return type Module

extra_repr() → str
Set the extra representation of the module

To print customized extra information, you should re-implement this method in your own modules. Both single-line and multi-line strings are acceptable.

fit(*X: numpy.ndarray*, *T: numpy.ndarray*, *E: numpy.ndarray*) → *synthcity.plugins.core.models.time_to_event.tte_date.TimeEventGAN*

float() → torch.nn.modules.module.T
Casts all floating point parameters and buffers to `float` datatype.

Note: This method modifies the module in-place.

Returns self

Return type Module

forward(*X: torch.Tensor*) → torch.Tensor
Defines the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

---

generate(*X: numpy.ndarray*) → numpy.ndarray

get_buffer(*target: str*) → torch.Tensor
Returns the buffer given by `target` if it exists, otherwise throws an error.

See the docstring for `get_submodule` for a more detailed explanation of this method's functionality as well as how to correctly specify `target`.

Parameters **target** – The fully-qualified string name of the buffer to look for. (See `get_submodule` for how to specify a fully-qualified string.)

Returns The buffer referenced by `target`

Return type torch.Tensor

Raises **AttributeError** – If the target string references an invalid path or resolves to something that is not a buffer

get_extra_state() → Any
Returns any extra state to include in the module's state_dict. Implement this and a corresponding *set_extra_state()* for your module if you need to store extra state. This function is called when building the module's *state_dict()*.

Note that extra state should be pickleable to ensure working serialization of the state_dict. We only provide provide backwards compatibility guarantees for serializing Tensors; other objects may break backwards compatibility if their serialized pickled form changes.

Returns Any extra state to store in the module's state_dict

Return type object

get_parameter(*target: str*) → torch.nn.parameter.Parameter
Returns the parameter given by `target` if it exists, otherwise throws an error.

See the docstring for `get_submodule` for a more detailed explanation of this method's functionality as well as how to correctly specify `target`.

Parameters **target** – The fully-qualified string name of the Parameter to look for. (See `get_submodule` for how to specify a fully-qualified string.)

Returns The Parameter referenced by `target`

Return type torch.nn.Parameter

Raises **AttributeError** – If the target string references an invalid path or resolves to something that is not an `nn.Parameter`

get_submodule(*target: str*) → torch.nn.modules.module.Module
Returns the submodule given by `target` if it exists, otherwise throws an error.

For example, let's say you have an `nn.Module` `A` that looks like this:

```
A(
    (net_b): Module(
        (net_c): Module(
            (conv): Conv2d(16, 33, kernel_size=(3, 3), stride=(2, 2))
        )
        (linear): Linear(in_features=100, out_features=200, bias=True)
    )
)
```

(The diagram shows an `nn.Module` `A`. `A` has a nested submodule `net_b`, which itself has two submodules `net_c` and `linear`. `net_c` then has a submodule `conv`.)

To check whether or not we have the `linear` submodule, we would call `get_submodule("net_b.linear")`. To check whether we have the `conv` submodule, we would call `get_submodule("net_b.net_c.conv")`.

The runtime of `get_submodule` is bounded by the degree of module nesting in `target`. A query against `named_modules` achieves the same result, but it is O(N) in the number of transitive modules. So, for a simple check to see if some submodule exists, `get_submodule` should always be used.

Parameters **target** – The fully-qualified string name of the submodule to look for. (See above example for how to specify a fully-qualified string.)

Returns The submodule referenced by `target`

Return type torch.nn.Module

Raises **AttributeError** – If the target string references an invalid path or resolves to something that is not an `nn.Module`

half() → torch.nn.modules.module.T
Casts all floating point parameters and buffers to `half` datatype.

Note: This method modifies the module in-place.

Returns self

Return type Module

ipu(*device: Optional[Union[int, torch.device]] = None*) → torch.nn.modules.module.T
Moves all model parameters and buffers to the IPU.

This also makes associated parameters and buffers different objects. So it should be called before constructing optimizer if the module will live on IPU while being optimized.

Note: This method modifies the module in-place.

Parameters **device** (*int, optional*) – if specified, all parameters will be copied to that device

Returns self

Return type Module

load_state_dict(*state_dict: Mapping[str, Any]*, *strict: bool = True*)
Copies parameters and buffers from *state_dict* into this module and its descendants. If `strict` is `True`, then the keys of *state_dict* must exactly match the keys returned by this module's `state_dict()` function.

Parameters

- **state_dict** (*dict*) – a dict containing parameters and persistent buffers.
- **strict** (*bool, optional*) – whether to strictly enforce that the keys in *state_dict* match the keys returned by this module's `state_dict()` function. Default: `True`

Returns

- **missing_keys** is a list of str containing the missing keys
- **unexpected_keys** is a list of str containing the unexpected keys

Return type `NamedTuple` with `missing_keys` and `unexpected_keys` fields

Note: If a parameter or buffer is registered as `None` and its corresponding key exists in *state_dict*, *load_state_dict()* will raise a `RuntimeError`.

---

modules() → Iterator[torch.nn.modules.module.Module]
Returns an iterator over all modules in the network.

Yields *Module* – a module in the network

Note: Duplicate modules are returned only once. In the following example, `l` will be returned only once.

Example:

```
>>> l = nn.Linear(2, 2)
>>> net = nn.Sequential(l, l)
>>> for idx, m in enumerate(net.modules()):
...     print(idx, '->', m)

0 -> Sequential(
  (0): Linear(in_features=2, out_features=2, bias=True)
  (1): Linear(in_features=2, out_features=2, bias=True)
```

(continues on next page)

(continued from previous page)

```
)
1 -> Linear(in_features=2, out_features=2, bias=True)
```

named_buffers(*prefix: str = ''*, *recurse: bool = True*) → Iterator[Tuple[str, torch.Tensor]]

Returns an iterator over module buffers, yielding both the name of the buffer as well as the buffer itself.

Parameters

- **prefix** (*str*) – prefix to prepend to all buffer names.
- **recurse** (*bool*) – if True, then yields buffers of this module and all submodules. Otherwise, yields only buffers that are direct members of this module.

Yields *(str, torch.Tensor)* – Tuple containing the name and buffer

Example:

```
>>> # xdoctest: +SKIP("undefined vars")
>>> for name, buf in self.named_buffers():
>>>    if name in ['running_var']:
>>>        print(buf.size())
```

named_children() → Iterator[Tuple[str, torch.nn.modules.module.Module]]

Returns an iterator over immediate children modules, yielding both the name of the module as well as the module itself.

Yields *(str, Module)* – Tuple containing a name and child module

Example:

```
>>> # xdoctest: +SKIP("undefined vars")
>>> for name, module in model.named_children():
>>>     if name in ['conv4', 'conv5']:
>>>         print(module)
```

named_modules(*memo: Optional[Set[torch.nn.modules.module.Module]] = None*, *prefix: str = ''*, *remove_duplicate: bool = True*)

Returns an iterator over all modules in the network, yielding both the name of the module as well as the module itself.

Parameters

- **memo** – a memo to store the set of modules already added to the result
- **prefix** – a prefix that will be added to the name of the module
- **remove_duplicate** – whether to remove the duplicated module instances in the result or not

Yields *(str, Module)* – Tuple of name and module

Note: Duplicate modules are returned only once. In the following example, `l` will be returned only once.

Example:

```
>>> l = nn.Linear(2, 2)
>>> net = nn.Sequential(l, l)
```

(continues on next page)

(continued from previous page)

```
>>> for idx, m in enumerate(net.named_modules()):
...     print(idx, '->', m)

0 -> ('', Sequential(
  (0): Linear(in_features=2, out_features=2, bias=True)
  (1): Linear(in_features=2, out_features=2, bias=True)
))
1 -> ('0', Linear(in_features=2, out_features=2, bias=True))
```

named_parameters(*prefix: str = ''*, *recurse: bool = True*) → Iterator[Tuple[str, torch.nn.parameter.Parameter]]

Returns an iterator over module parameters, yielding both the name of the parameter as well as the parameter itself.

Parameters

- **prefix** (*str*) – prefix to prepend to all parameter names.
- **recurse** (*bool*) – if True, then yields parameters of this module and all submodules. Otherwise, yields only parameters that are direct members of this module.

Yields *(str, Parameter)* – Tuple containing the name and parameter

Example:

```
>>> # xdoctest: +SKIP("undefined vars")
>>> for name, param in self.named_parameters():
>>>    if name in ['bias']:
>>>        print(param.size())
```

parameters(*recurse: bool = True*) → Iterator[torch.nn.parameter.Parameter]

Returns an iterator over module parameters.

This is typically passed to an optimizer.

Parameters **recurse** (*bool*) – if True, then yields parameters of this module and all submodules. Otherwise, yields only parameters that are direct members of this module.

Yields *Parameter* – module parameter

Example:

```
>>> # xdoctest: +SKIP("undefined vars")
>>> for param in model.parameters():
>>>     print(type(param), param.size())
<class 'torch.Tensor'> (20L,)
<class 'torch.Tensor'> (20L, 1L, 5L, 5L)
```

register_backward_hook(*hook: Callable[[torch.nn.modules.module.Module, Union[Tuple[torch.Tensor, ...], torch.Tensor], Union[Tuple[torch.Tensor, ...], torch.Tensor]], Union[None, torch.Tensor]]*) → torch.utils.hooks.RemovableHandle

Registers a backward hook on the module.

This function is deprecated in favor of `register_full_backward_hook()` and the behavior of this function will change in future versions.

Returns a handle that can be used to remove the added hook by calling `handle.remove()`

Return type `torch.utils.hooks.RemovableHandle`

register_buffer(*name: str*, *tensor: Optional[torch.Tensor]*, *persistent: bool = True*) → None
Adds a buffer to the module.

This is typically used to register a buffer that should not to be considered a model parameter. For example, BatchNorm's `running_mean` is not a parameter, but is part of the module's state. Buffers, by default, are persistent and will be saved alongside parameters. This behavior can be changed by setting `persistent` to `False`. The only difference between a persistent buffer and a non-persistent buffer is that the latter will not be a part of this module's *state_dict*.

Buffers can be accessed as attributes using given names.

Parameters

- **name** (*str*) – name of the buffer. The buffer can be accessed from this module using the given name
- **tensor** (*Tensor or None*) – buffer to be registered. If `None`, then operations that run on buffers, such as *cuda*, are ignored. If `None`, the buffer is **not** included in the module's *state_dict*.
- **persistent** (*bool*) – whether the buffer is part of this module's *state_dict*.

Example:

```
>>> # xdoctest: +SKIP("undefined vars")
>>> self.register_buffer('running_mean', torch.zeros(num_features))
```

register_forward_hook(*hook: Callable[[...], None]*) → torch.utils.hooks.RemovableHandle
Registers a forward hook on the module.

The hook will be called every time after *forward()* has computed an output. It should have the following signature:

```
hook(module, input, output) -> None or modified output
```

The input contains only the positional arguments given to the module. Keyword arguments won't be passed to the hooks and only to the `forward`. The hook can modify the output. It can modify the input inplace but it will not have effect on forward since this is called after *forward()* is called.

Returns a handle that can be used to remove the added hook by calling `handle.remove()`

Return type `torch.utils.hooks.RemovableHandle`

register_forward_pre_hook(*hook: Callable[[...], None]*) → torch.utils.hooks.RemovableHandle
Registers a forward pre-hook on the module.

The hook will be called every time before *forward()* is invoked. It should have the following signature:

```
hook(module, input) -> None or modified input
```

The input contains only the positional arguments given to the module. Keyword arguments won't be passed to the hooks and only to the `forward`. The hook can modify the input. User can either return a tuple or a single modified value in the hook. We will wrap the value into a tuple if a single value is returned(unless that value is already a tuple).

Returns a handle that can be used to remove the added hook by calling `handle.remove()`

Return type `torch.utils.hooks.RemovableHandle`

register_full_backward_hook(*hook: Callable[[torch.nn.modules.module.Module, Union[Tuple[torch.Tensor, ...], torch.Tensor], Union[Tuple[torch.Tensor, ...], torch.Tensor]], Union[None, torch.Tensor]]*) → torch.utils.hooks.RemovableHandle

Registers a backward hook on the module.

The hook will be called every time the gradients with respect to a module are computed, i.e. the hook will execute if and only if the gradients with respect to module outputs are computed. The hook should have the following signature:

```
hook(module, grad_input, grad_output) -> tuple(Tensor) or None
```

The `grad_input` and `grad_output` are tuples that contain the gradients with respect to the inputs and outputs respectively. The hook should not modify its arguments, but it can optionally return a new gradient with respect to the input that will be used in place of `grad_input` in subsequent computations. `grad_input` will only correspond to the inputs given as positional arguments and all kwarg arguments are ignored. Entries in `grad_input` and `grad_output` will be `None` for all non-Tensor arguments.

For technical reasons, when this hook is applied to a Module, its forward function will receive a view of each Tensor passed to the Module. Similarly the caller will receive a view of each Tensor returned by the Module's forward function.

Warning: Modifying inputs or outputs inplace is not allowed when using backward hooks and will raise an error.

Returns a handle that can be used to remove the added hook by calling `handle.remove()`

Return type `torch.utils.hooks.RemovableHandle`

register_load_state_dict_post_hook(*hook*)

Registers a post hook to be run after module's `load_state_dict` is called.

It should have the following signature:: hook(module, incompatible_keys) -> None

The `module` argument is the current module that this hook is registered on, and the `incompatible_keys` argument is a `NamedTuple` consisting of attributes `missing_keys` and `unexpected_keys`. `missing_keys` is a `list` of `str` containing the missing keys and `unexpected_keys` is a `list` of `str` containing the unexpected keys.

The given incompatible_keys can be modified inplace if needed.

Note that the checks performed when calling *load_state_dict()* with `strict=True` are affected by modifications the hook makes to `missing_keys` or `unexpected_keys`, as expected. Additions to either set of keys will result in an error being thrown when `strict=True`, and clearning out both missing and unexpected keys will avoid an error.

Returns a handle that can be used to remove the added hook by calling `handle.remove()`

Return type `torch.utils.hooks.RemovableHandle`

register_module(*name: str*, *module: Optional[torch.nn.modules.module.Module]*) → None

Alias for *add_module()*.

register_parameter(*name: str*, *param: Optional[torch.nn.parameter.Parameter]*) → None

Adds a parameter to the module.

The parameter can be accessed as an attribute using given name.

Parameters

- **name** (*str*) – name of the parameter. The parameter can be accessed from this module using the given name
- **param** (*Parameter or None*) – parameter to be added to the module. If `None`, then operations that run on parameters, such as *cuda*, are ignored. If `None`, the parameter is **not** included in the module's *state_dict*.

requires_grad_(*requires_grad: bool = True*) → torch.nn.modules.module.T
Change if autograd should record operations on parameters in this module.

This method sets the parameters' `requires_grad` attributes in-place.

This method is helpful for freezing part of the module for finetuning or training parts of a model individually (e.g., GAN training).

See locally-disable-grad-doc for a comparison between *.requires_grad_()* and several similar mechanisms that may be confused with it.

Parameters **requires_grad** (*bool*) – whether autograd should record operations on parameters in this module. Default: `True`.

Returns self

Return type Module

set_extra_state(*state: Any*)
This function is called from *load_state_dict()* to handle any extra state found within the *state_dict*. Implement this function and a corresponding *get_extra_state()* for your module if you need to store extra state within its *state_dict*.

Parameters **state** (*dict*) – Extra state from the *state_dict*

share_memory() → torch.nn.modules.module.T
See `torch.Tensor.share_memory_()`

state_dict(**args*, *destination=None*, *prefix=''*, *keep_vars=False*)
Returns a dictionary containing references to the whole state of the module.

Both parameters and persistent buffers (e.g. running averages) are included. Keys are corresponding parameter and buffer names. Parameters and buffers set to `None` are not included.

Note: The returned object is a shallow copy. It contains references to the module's parameters and buffers.

Warning: Currently `state_dict()` also accepts positional arguments for `destination`, `prefix` and `keep_vars` in order. However, this is being deprecated and keyword arguments will be enforced in future releases.

Warning: Please avoid the use of argument `destination` as it is not designed for end-users.

Parameters

- **destination** (*dict, optional*) – If provided, the state of module will be updated into the dict and the same object is returned. Otherwise, an `OrderedDict` will be created and returned. Default: `None`.
- **prefix** (*str, optional*) – a prefix added to parameter and buffer names to compose the keys in state_dict. Default: `''`.

- **keep_vars** (*bool, optional*) – by default the `Tensor` s returned in the state dict are detached from autograd. If it's set to `True`, detaching will not be performed. Default: `False`.

Returns a dictionary containing a whole state of the module

Return type dict

Example:

```
>>> # xdoctest: +SKIP("undefined vars")
>>> module.state_dict().keys()
['bias', 'weight']
```

to(**args*, ***kwargs*)

Moves and/or casts the parameters and buffers.

This can be called as

to(*device=None*, *dtype=None*, *non_blocking=False*)

to(*dtype*, *non_blocking=False*)

to(*tensor*, *non_blocking=False*)

to(*memory_format=torch.channels_last*)

Its signature is similar to `torch.Tensor.to()`, but only accepts floating point or complex `dtypes`. In addition, this method will only cast the floating point or complex parameters and buffers to `dtype` (if given). The integral parameters and buffers will be moved `device`, if that is given, but with dtypes unchanged. When `non_blocking` is set, it tries to convert/move asynchronously with respect to the host if possible, e.g., moving CPU Tensors with pinned memory to CUDA devices.

See below for examples.

Note: This method modifies the module in-place.

Parameters

- **device** (`torch.device`) – the desired device of the parameters and buffers in this module
- **dtype** (`torch.dtype`) – the desired floating point or complex dtype of the parameters and buffers in this module
- **tensor** (*torch.Tensor*) – Tensor whose dtype and device are the desired dtype and device for all parameters and buffers in this module
- **memory_format** (`torch.memory_format`) – the desired memory format for 4D parameters and buffers in this module (keyword only argument)

Returns self

Return type Module

Examples:

```
>>> # xdoctest: +IGNORE_WANT("non-deterministic")
>>> linear = nn.Linear(2, 2)
>>> linear.weight
Parameter containing:
tensor([[ 0.1913, -0.3420],
        [-0.5113, -0.2325]])
>>> linear.to(torch.double)
Linear(in_features=2, out_features=2, bias=True)
>>> linear.weight
Parameter containing:
tensor([[ 0.1913, -0.3420],
        [-0.5113, -0.2325]], dtype=torch.float64)
>>> # xdoctest: +REQUIRES(env:TORCH_DOCTEST_CUDA1)
>>> gpu1 = torch.device("cuda:1")
>>> linear.to(gpu1, dtype=torch.half, non_blocking=True)
Linear(in_features=2, out_features=2, bias=True)
>>> linear.weight
Parameter containing:
tensor([[ 0.1914, -0.3420],
        [-0.5112, -0.2324]], dtype=torch.float16, device='cuda:1')
>>> cpu = torch.device("cpu")
>>> linear.to(cpu)
Linear(in_features=2, out_features=2, bias=True)
>>> linear.weight
Parameter containing:
tensor([[ 0.1914, -0.3420],
        [-0.5112, -0.2324]], dtype=torch.float16)

>>> linear = nn.Linear(2, 2, bias=None).to(torch.cdouble)
>>> linear.weight
Parameter containing:
tensor([[ 0.3741+0.j,  0.2382+0.j],
        [ 0.5593+0.j, -0.4443+0.j]], dtype=torch.complex128)
>>> linear(torch.ones(3, 2, dtype=torch.cdouble))
tensor([[0.6122+0.j, 0.1150+0.j],
        [0.6122+0.j, 0.1150+0.j],
        [0.6122+0.j, 0.1150+0.j]], dtype=torch.complex128)
```

to_empty(*, *device: Union[str, torch.device]*) → torch.nn.modules.module.T
Moves the parameters and buffers to the specified device without copying storage.

Parameters **device** (torch.device) – The desired device of the parameters and buffers in this module.

Returns self

Return type Module

train(*mode: bool = True*) → torch.nn.modules.module.T
Sets the module in training mode.

This has any effect only on certain modules. See documentations of particular modules for details of their behaviors in training/evaluation mode, if they are affected, e.g. Dropout, BatchNorm, etc.

Parameters **mode** (*bool*) – whether to set training mode (True) or evaluation mode (False). Default: True.

Returns self

Return type Module

training: bool

type(*dst_type: Union[torch.dtype, str]*) → torch.nn.modules.module.T
Casts all parameters and buffers to `dst_type`.

Note: This method modifies the module in-place.

Parameters **dst_type** (*type or string*) – the desired type

Returns self

Return type Module

xpu(*device: Optional[Union[int, torch.device]] = None*) → torch.nn.modules.module.T
Moves all model parameters and buffers to the XPU.

This also makes associated parameters and buffers different objects. So it should be called before constructing optimizer if the module will live on XPU while being optimized.

Note: This method modifies the module in-place.

Parameters **device** (*int, optional*) – if specified, all parameters will be copied to that device

Returns self

Return type Module

zero_grad(*set_to_none: bool = False*) → None
Sets gradients of all model parameters to zero. See similar function under `torch.optim.Optimizer` for more context.

Parameters **set_to_none** (*bool*) – instead of setting to zero, set the grads to None. See `torch.optim.Optimizer.zero_grad()` for details.

synthcity.plugins.core.models.time_to_event.tte_survival_function_regression module

class SurvivalFunctionTimeToEvent(*time_points: int = 100*, *model_search_n_iter: Optional[int] = None*, *device: Any = device(type='cpu')*, ***kwargs: Any*)
Bases: `synthcity.plugins.core.models.time_to_event._base.TimeToEventPlugin`

fit(*X: pandas.core.frame.DataFrame*, *T: pandas.core.series.Series*, *Y: pandas.core.series.Series*) → synthcity.plugins.core.models.time_to_event._base.TimeToEventPlugin
Training logic

static hyperparameter_space(***kwargs: Any*) → List[*synthcity.plugins.core.distribution.Distribution*]
Returns the hyperparameter space for the derived plugin.

static load(*buff: bytes*) → Any

static load_dict(*representation: dict*) → Any

static name() → str
The name of the plugin.

predict(*X: pandas.core.frame.DataFrame*) → pandas.core.series.Series
Predict time-to-event

predict_any(*X: pandas.core.frame.DataFrame*, *E: pandas.core.series.Series*) → pandas.core.series.Series
Predict time-to-event or censoring

classmethod sample_hyperparameters(**args: Any*, ***kwargs: Any*) → Dict[str, Any]
Sample value from the hyperparameter space for the current plugin.

save() → bytes

save_dict() → dict

save_to_file(*path: pathlib.Path*) → bytes

static version() → str
API version

12.1.6 Images

synthcity.plugins.core.models.convnet module

class ConditionalDiscriminator(*model: torch.nn.modules.module.Module*, *n_channels: int*, *height: int*, *width: int*, *cond: Optional[torch.Tensor] = None*, *cond_embedding_n_units_hidden: int = 100*, *device: Any = device(type='cpu')*)
Bases: torch.nn.modules.module.Module

Parameters

- **model** – nn.Module Core model.
- **n_channels** – int Number of channels in images
- **height** – int Image height
- **width** – int Image width
- **cond** – torch.Tensor The reference conditional
- **cond_embedding_n_units_hidden** – int Size of the conditional embedding layer

T_destination
alias of TypeVar('T_destination', bound=Dict[str, Any])

add_module(*name: str*, *module: Optional[torch.nn.modules.module.Module]*) → None
Adds a child module to the current module.

The module can be accessed as an attribute using the given name.

Parameters

- **name** (*str*) – name of the child module. The child module can be accessed from this module using the given name
- **module** (*Module*) – child module to be added to the module.

apply(*fn: Callable[[torch.nn.modules.module.Module], None]*) → torch.nn.modules.module.T
Applies fn recursively to every submodule (as returned by .children()) as well as self. Typical use includes initializing the parameters of a model (see also nn-init-doc).

Parameters **fn** (Module -> None) – function to be applied to each submodule

Returns self

Return type Module

Example:

```
>>> @torch.no_grad()
>>> def init_weights(m):
>>>     print(m)
>>>     if type(m) == nn.Linear:
>>>         m.weight.fill_(1.0)
>>>         print(m.weight)
>>> net = nn.Sequential(nn.Linear(2, 2), nn.Linear(2, 2))
>>> net.apply(init_weights)
Linear(in_features=2, out_features=2, bias=True)
Parameter containing:
tensor([[1., 1.],
        [1., 1.]], requires_grad=True)
Linear(in_features=2, out_features=2, bias=True)
Parameter containing:
tensor([[1., 1.],
        [1., 1.]], requires_grad=True)
Sequential(
  (0): Linear(in_features=2, out_features=2, bias=True)
  (1): Linear(in_features=2, out_features=2, bias=True)
)
```

bfloat16() → torch.nn.modules.module.T

Casts all floating point parameters and buffers to `bfloat16` datatype.

Note: This method modifies the module in-place.

Returns self

Return type Module

buffers(*recurse: bool = True*) → Iterator[torch.Tensor]

Returns an iterator over module buffers.

Parameters **recurse** (*bool*) – if True, then yields buffers of this module and all submodules. Otherwise, yields only buffers that are direct members of this module.

Yields *torch.Tensor* – module buffer

Example:

```
>>> # xdoctest: +SKIP("undefined vars")
>>> for buf in model.buffers():
>>>     print(type(buf), buf.size())
<class 'torch.Tensor'> (20L,)
<class 'torch.Tensor'> (20L, 1L, 5L, 5L)
```

children() → Iterator[torch.nn.modules.module.Module]

Returns an iterator over immediate children modules.

Yields *Module* – a child module

cpu() → torch.nn.modules.module.T
Moves all model parameters and buffers to the CPU.

Note: This method modifies the module in-place.

Returns self

Return type Module

cuda(*device: Optional[Union[int, torch.device]] = None*) → torch.nn.modules.module.T
Moves all model parameters and buffers to the GPU.

This also makes associated parameters and buffers different objects. So it should be called before constructing optimizer if the module will live on GPU while being optimized.

Note: This method modifies the module in-place.

Parameters **device** (*int, optional*) – if specified, all parameters will be copied to that device

Returns self

Return type Module

double() → torch.nn.modules.module.T
Casts all floating point parameters and buffers to `double` datatype.

Note: This method modifies the module in-place.

Returns self

Return type Module

dump_patches: bool = False

eval() → torch.nn.modules.module.T
Sets the module in evaluation mode.

This has any effect only on certain modules. See documentations of particular modules for details of their behaviors in training/evaluation mode, if they are affected, e.g. `Dropout`, `BatchNorm`, etc.

This is equivalent with `self.train(False)`.

See locally-disable-grad-doc for a comparison between *.eval()* and several similar mechanisms that may be confused with it.

Returns self

Return type Module

extra_repr() → str
Set the extra representation of the module

To print customized extra information, you should re-implement this method in your own modules. Both single-line and multi-line strings are acceptable.

float() → torch.nn.modules.module.T
Casts all floating point parameters and buffers to `float` datatype.

Note: This method modifies the module in-place.

Returns self

Return type Module

forward(*X: torch.Tensor*, *cond: Optional[torch.Tensor] = None*) → torch.Tensor
Defines the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

---

get_buffer(*target: str*) → torch.Tensor
Returns the buffer given by `target` if it exists, otherwise throws an error.

See the docstring for `get_submodule` for a more detailed explanation of this method's functionality as well as how to correctly specify `target`.

Parameters **target** – The fully-qualified string name of the buffer to look for. (See `get_submodule` for how to specify a fully-qualified string.)

Returns The buffer referenced by `target`

Return type torch.Tensor

Raises **AttributeError** – If the target string references an invalid path or resolves to something that is not a buffer

get_extra_state() → Any
Returns any extra state to include in the module's state_dict. Implement this and a corresponding *set_extra_state()* for your module if you need to store extra state. This function is called when building the module's *state_dict()*.

Note that extra state should be pickleable to ensure working serialization of the state_dict. We only provide provide backwards compatibility guarantees for serializing Tensors; other objects may break backwards compatibility if their serialized pickled form changes.

Returns Any extra state to store in the module's state_dict

Return type object

get_parameter(*target: str*) → torch.nn.parameter.Parameter
Returns the parameter given by `target` if it exists, otherwise throws an error.

See the docstring for `get_submodule` for a more detailed explanation of this method's functionality as well as how to correctly specify `target`.

Parameters **target** – The fully-qualified string name of the Parameter to look for. (See `get_submodule` for how to specify a fully-qualified string.)

Returns The Parameter referenced by `target`

Return type torch.nn.Parameter

Raises `AttributeError` – If the target string references an invalid path or resolves to something that is not an `nn.Parameter`

`get_submodule`(*target: str*) → torch.nn.modules.module.Module
Returns the submodule given by `target` if it exists, otherwise throws an error.

For example, let's say you have an `nn.Module` `A` that looks like this:

```
A(
    (net_b): Module(
        (net_c): Module(
            (conv): Conv2d(16, 33, kernel_size=(3, 3), stride=(2, 2))
        )
        (linear): Linear(in_features=100, out_features=200, bias=True)
    )
)
```

(The diagram shows an `nn.Module` `A`. `A` has a nested submodule `net_b`, which itself has two submodules `net_c` and `linear`. `net_c` then has a submodule `conv`.)

To check whether or not we have the `linear` submodule, we would call `get_submodule("net_b.linear")`. To check whether we have the `conv` submodule, we would call `get_submodule("net_b.net_c.conv")`.

The runtime of `get_submodule` is bounded by the degree of module nesting in `target`. A query against `named_modules` achieves the same result, but it is O(N) in the number of transitive modules. So, for a simple check to see if some submodule exists, `get_submodule` should always be used.

Parameters `target` – The fully-qualified string name of the submodule to look for. (See above example for how to specify a fully-qualified string.)

Returns The submodule referenced by `target`

Return type torch.nn.Module

Raises `AttributeError` – If the target string references an invalid path or resolves to something that is not an `nn.Module`

`half`() → torch.nn.modules.module.T
Casts all floating point parameters and buffers to `half` datatype.

Note: This method modifies the module in-place.

Returns self

Return type Module

`ipu`(*device: Optional[Union[int, torch.device]] = None*) → torch.nn.modules.module.T
Moves all model parameters and buffers to the IPU.

This also makes associated parameters and buffers different objects. So it should be called before constructing optimizer if the module will live on IPU while being optimized.

Note: This method modifies the module in-place.

Parameters **device** (*int, optional*) – if specified, all parameters will be copied to that device

Returns self

Return type Module

load_state_dict(*state_dict: Mapping[str, Any]*, *strict: bool = True*)

Copies parameters and buffers from *state_dict* into this module and its descendants. If `strict` is `True`, then the keys of *state_dict* must exactly match the keys returned by this module's `state_dict()` function.

Parameters

- **state_dict** (*dict*) – a dict containing parameters and persistent buffers.
- **strict** (*bool, optional*) – whether to strictly enforce that the keys in *state_dict* match the keys returned by this module's `state_dict()` function. Default: `True`

Returns

- **missing_keys** is a list of str containing the missing keys
- **unexpected_keys** is a list of str containing the unexpected keys

Return type `NamedTuple` with `missing_keys` and `unexpected_keys` fields

Note: If a parameter or buffer is registered as `None` and its corresponding key exists in *state_dict*, *load_state_dict()* will raise a `RuntimeError`.

---

modules() → Iterator[torch.nn.modules.module.Module]

Returns an iterator over all modules in the network.

Yields *Module* – a module in the network

Note: Duplicate modules are returned only once. In the following example, `l` will be returned only once.

Example:

```
>>> l = nn.Linear(2, 2)
>>> net = nn.Sequential(l, l)
>>> for idx, m in enumerate(net.modules()):
...     print(idx, '->', m)

0 -> Sequential(
  (0): Linear(in_features=2, out_features=2, bias=True)
  (1): Linear(in_features=2, out_features=2, bias=True)
)
1 -> Linear(in_features=2, out_features=2, bias=True)
```

named_buffers(*prefix: str = ''*, *recurse: bool = True*) → Iterator[Tuple[str, torch.Tensor]]

Returns an iterator over module buffers, yielding both the name of the buffer as well as the buffer itself.

Parameters

- **prefix** (*str*) – prefix to prepend to all buffer names.
- **recurse** (*bool*) – if True, then yields buffers of this module and all submodules. Otherwise, yields only buffers that are direct members of this module.

Yields *(str, torch.Tensor)* – Tuple containing the name and buffer

Example:

```
>>> # xdoctest: +SKIP("undefined vars")
>>> for name, buf in self.named_buffers():
>>>    if name in ['running_var']:
>>>        print(buf.size())
```

named_children() → Iterator[Tuple[str, torch.nn.modules.module.Module]]

Returns an iterator over immediate children modules, yielding both the name of the module as well as the module itself.

Yields *(str, Module)* – Tuple containing a name and child module

Example:

```
>>> # xdoctest: +SKIP("undefined vars")
>>> for name, module in model.named_children():
>>>     if name in ['conv4', 'conv5']:
>>>         print(module)
```

named_modules(*memo: Optional[Set[torch.nn.modules.module.Module]] = None*, *prefix: str = ''*, *remove_duplicate: bool = True*)

Returns an iterator over all modules in the network, yielding both the name of the module as well as the module itself.

Parameters

- **memo** – a memo to store the set of modules already added to the result
- **prefix** – a prefix that will be added to the name of the module
- **remove_duplicate** – whether to remove the duplicated module instances in the result or not

Yields *(str, Module)* – Tuple of name and module

Note: Duplicate modules are returned only once. In the following example, `l` will be returned only once.

Example:

```
>>> l = nn.Linear(2, 2)
>>> net = nn.Sequential(l, l)
>>> for idx, m in enumerate(net.named_modules()):
...     print(idx, '->', m)

0 -> ('', Sequential(
  (0): Linear(in_features=2, out_features=2, bias=True)
  (1): Linear(in_features=2, out_features=2, bias=True)
))
1 -> ('0', Linear(in_features=2, out_features=2, bias=True))
```

named_parameters(*prefix: str = ''*, *recurse: bool = True*) → Iterator[Tuple[str, torch.nn.parameter.Parameter]]

Returns an iterator over module parameters, yielding both the name of the parameter as well as the parameter itself.

Parameters

- **prefix** (*str*) – prefix to prepend to all parameter names.
- **recurse** (*bool*) – if True, then yields parameters of this module and all submodules. Otherwise, yields only parameters that are direct members of this module.

Yields *(str, Parameter)* – Tuple containing the name and parameter

Example:

```
>>> # xdoctest: +SKIP("undefined vars")
>>> for name, param in self.named_parameters():
>>>    if name in ['bias']:
>>>        print(param.size())
```

parameters(*recurse: bool = True*) → Iterator[torch.nn.parameter.Parameter]

Returns an iterator over module parameters.

This is typically passed to an optimizer.

Parameters **recurse** (*bool*) – if True, then yields parameters of this module and all submodules. Otherwise, yields only parameters that are direct members of this module.

Yields *Parameter* – module parameter

Example:

```
>>> # xdoctest: +SKIP("undefined vars")
>>> for param in model.parameters():
>>>     print(type(param), param.size())
<class 'torch.Tensor'> (20L,)
<class 'torch.Tensor'> (20L, 1L, 5L, 5L)
```

register_backward_hook(*hook: Callable[[torch.nn.modules.module.Module, Union[Tuple[torch.Tensor, ...], torch.Tensor], Union[Tuple[torch.Tensor, ...], torch.Tensor]], Union[None, torch.Tensor]]*) → torch.utils.hooks.RemovableHandle

Registers a backward hook on the module.

This function is deprecated in favor of `register_full_backward_hook()` and the behavior of this function will change in future versions.

Returns a handle that can be used to remove the added hook by calling `handle.remove()`

Return type `torch.utils.hooks.RemovableHandle`

register_buffer(*name: str*, *tensor: Optional[torch.Tensor]*, *persistent: bool = True*) → None

Adds a buffer to the module.

This is typically used to register a buffer that should not to be considered a model parameter. For example, BatchNorm's `running_mean` is not a parameter, but is part of the module's state. Buffers, by default, are persistent and will be saved alongside parameters. This behavior can be changed by setting `persistent` to `False`. The only difference between a persistent buffer and a non-persistent buffer is that the latter will not be a part of this module's *state_dict*.

Buffers can be accessed as attributes using given names.

Parameters

- **name** (*str*) – name of the buffer. The buffer can be accessed from this module using the given name

- **tensor** (*Tensor or None*) – buffer to be registered. If None, then operations that run on buffers, such as *cuda*, are ignored. If None, the buffer is **not** included in the module's *state_dict*.
- **persistent** (*bool*) – whether the buffer is part of this module's *state_dict*.

Example:

```
>>> # xdoctest: +SKIP("undefined vars")
>>> self.register_buffer('running_mean', torch.zeros(num_features))
```

register_forward_hook(*hook: Callable[[...], None]*) → torch.utils.hooks.RemovableHandle

Registers a forward hook on the module.

The hook will be called every time after *forward()* has computed an output. It should have the following signature:

```
hook(module, input, output) -> None or modified output
```

The input contains only the positional arguments given to the module. Keyword arguments won't be passed to the hooks and only to the forward. The hook can modify the output. It can modify the input inplace but it will not have effect on forward since this is called after *forward()* is called.

Returns a handle that can be used to remove the added hook by calling handle.remove()

Return type torch.utils.hooks.RemovableHandle

register_forward_pre_hook(*hook: Callable[[...], None]*) → torch.utils.hooks.RemovableHandle

Registers a forward pre-hook on the module.

The hook will be called every time before *forward()* is invoked. It should have the following signature:

```
hook(module, input) -> None or modified input
```

The input contains only the positional arguments given to the module. Keyword arguments won't be passed to the hooks and only to the forward. The hook can modify the input. User can either return a tuple or a single modified value in the hook. We will wrap the value into a tuple if a single value is returned(unless that value is already a tuple).

Returns a handle that can be used to remove the added hook by calling handle.remove()

Return type torch.utils.hooks.RemovableHandle

register_full_backward_hook(*hook: Callable[[torch.nn.modules.module.Module, Union[Tuple[torch.Tensor, ...], torch.Tensor], Union[Tuple[torch.Tensor, ...], torch.Tensor]], Union[None, torch.Tensor]]*) → torch.utils.hooks.RemovableHandle

Registers a backward hook on the module.

The hook will be called every time the gradients with respect to a module are computed, i.e. the hook will execute if and only if the gradients with respect to module outputs are computed. The hook should have the following signature:

```
hook(module, grad_input, grad_output) -> tuple(Tensor) or None
```

The grad_input and grad_output are tuples that contain the gradients with respect to the inputs and outputs respectively. The hook should not modify its arguments, but it can optionally return a new gradient with respect to the input that will be used in place of grad_input in subsequent computations. grad_input will only correspond to the inputs given as positional arguments and all kwarg arguments are ignored. Entries in grad_input and grad_output will be None for all non-Tensor arguments.

For technical reasons, when this hook is applied to a Module, its forward function will receive a view of each Tensor passed to the Module. Similarly the caller will receive a view of each Tensor returned by the Module's forward function.

Warning: Modifying inputs or outputs inplace is not allowed when using backward hooks and will raise an error.

Returns a handle that can be used to remove the added hook by calling `handle.remove()`

Return type `torch.utils.hooks.RemovableHandle`

register_load_state_dict_post_hook(*hook*)

Registers a post hook to be run after module's `load_state_dict` is called.

It should have the following signature:: hook(module, incompatible_keys) -> None

The `module` argument is the current module that this hook is registered on, and the `incompatible_keys` argument is a `NamedTuple` consisting of attributes `missing_keys` and `unexpected_keys`. `missing_keys` is a `list` of `str` containing the missing keys and `unexpected_keys` is a `list` of `str` containing the unexpected keys.

The given incompatible_keys can be modified inplace if needed.

Note that the checks performed when calling *load_state_dict()* with `strict=True` are affected by modifications the hook makes to `missing_keys` or `unexpected_keys`, as expected. Additions to either set of keys will result in an error being thrown when `strict=True`, and clearning out both missing and unexpected keys will avoid an error.

Returns a handle that can be used to remove the added hook by calling `handle.remove()`

Return type `torch.utils.hooks.RemovableHandle`

register_module(*name: str*, *module: Optional[torch.nn.modules.module.Module]*) → None

Alias for *add_module()*.

register_parameter(*name: str*, *param: Optional[torch.nn.parameter.Parameter]*) → None

Adds a parameter to the module.

The parameter can be accessed as an attribute using given name.

Parameters

- **name** (*str*) – name of the parameter. The parameter can be accessed from this module using the given name
- **param** (*Parameter or None*) – parameter to be added to the module. If `None`, then operations that run on parameters, such as *cuda*, are ignored. If `None`, the parameter is **not** included in the module's *state_dict*.

requires_grad_(*requires_grad: bool = True*) → torch.nn.modules.module.T

Change if autograd should record operations on parameters in this module.

This method sets the parameters' `requires_grad` attributes in-place.

This method is helpful for freezing part of the module for finetuning or training parts of a model individually (e.g., GAN training).

See locally-disable-grad-doc for a comparison between *.requires_grad_()* and several similar mechanisms that may be confused with it.

Parameters **requires_grad** (*bool*) – whether autograd should record operations on parameters in this module. Default: `True`.

Returns self

Return type Module

set_extra_state(*state: Any*)
This function is called from *load_state_dict()* to handle any extra state found within the *state_dict*. Implement this function and a corresponding *get_extra_state()* for your module if you need to store extra state within its *state_dict*.

Parameters **state** (*dict*) – Extra state from the *state_dict*

share_memory() → torch.nn.modules.module.T
See `torch.Tensor.share_memory_()`

state_dict(**args*, *destination=None*, *prefix=''*, *keep_vars=False*)
Returns a dictionary containing references to the whole state of the module.

Both parameters and persistent buffers (e.g. running averages) are included. Keys are corresponding parameter and buffer names. Parameters and buffers set to `None` are not included.

Note: The returned object is a shallow copy. It contains references to the module's parameters and buffers.

Warning: Currently `state_dict()` also accepts positional arguments for `destination`, `prefix` and `keep_vars` in order. However, this is being deprecated and keyword arguments will be enforced in future releases.

Warning: Please avoid the use of argument `destination` as it is not designed for end-users.

Parameters

- **destination** (*dict, optional*) – If provided, the state of module will be updated into the dict and the same object is returned. Otherwise, an `OrderedDict` will be created and returned. Default: `None`.
- **prefix** (*str, optional*) – a prefix added to parameter and buffer names to compose the keys in state_dict. Default: `''`.
- **keep_vars** (*bool, optional*) – by default the `Tensor` s returned in the state dict are detached from autograd. If it's set to `True`, detaching will not be performed. Default: `False`.

Returns a dictionary containing a whole state of the module

Return type dict

Example:

```
>>> # xdoctest: +SKIP("undefined vars")
>>> module.state_dict().keys()
['bias', 'weight']
```

to(**args*, ***kwargs*)
Moves and/or casts the parameters and buffers.

This can be called as

to(*device=None*, *dtype=None*, *non_blocking=False*)

to(*dtype*, *non_blocking=False*)

to(*tensor*, *non_blocking=False*)

to(*memory_format=torch.channels_last*)

Its signature is similar to `torch.Tensor.to()`, but only accepts floating point or complex `dtype`s. In addition, this method will only cast the floating point or complex parameters and buffers to `dtype` (if given). The integral parameters and buffers will be moved `device`, if that is given, but with dtypes unchanged. When `non_blocking` is set, it tries to convert/move asynchronously with respect to the host if possible, e.g., moving CPU Tensors with pinned memory to CUDA devices.

See below for examples.

Note: This method modifies the module in-place.

Parameters

- **device** (`torch.device`) – the desired device of the parameters and buffers in this module
- **dtype** (`torch.dtype`) – the desired floating point or complex dtype of the parameters and buffers in this module
- **tensor** (*torch.Tensor*) – Tensor whose dtype and device are the desired dtype and device for all parameters and buffers in this module
- **memory_format** (`torch.memory_format`) – the desired memory format for 4D parameters and buffers in this module (keyword only argument)

Returns self

Return type Module

Examples:

```
>>> # xdoctest: +IGNORE_WANT("non-deterministic")
>>> linear = nn.Linear(2, 2)
>>> linear.weight
Parameter containing:
tensor([[ 0.1913, -0.3420],
        [-0.5113, -0.2325]])
>>> linear.to(torch.double)
Linear(in_features=2, out_features=2, bias=True)
>>> linear.weight
Parameter containing:
tensor([[ 0.1913, -0.3420],
        [-0.5113, -0.2325]], dtype=torch.float64)
>>> # xdoctest: +REQUIRES(env:TORCH_DOCTEST_CUDA1)
>>> gpu1 = torch.device("cuda:1")
>>> linear.to(gpu1, dtype=torch.half, non_blocking=True)
Linear(in_features=2, out_features=2, bias=True)
>>> linear.weight
Parameter containing:
tensor([[ 0.1914, -0.3420],
```

(continues on next page)

(continued from previous page)

```
        [-0.5112, -0.2324]], dtype=torch.float16, device='cuda:1')
>>> cpu = torch.device("cpu")
>>> linear.to(cpu)
Linear(in_features=2, out_features=2, bias=True)
>>> linear.weight
Parameter containing:
tensor([[ 0.1914, -0.3420],
        [-0.5112, -0.2324]], dtype=torch.float16)

>>> linear = nn.Linear(2, 2, bias=None).to(torch.cdouble)
>>> linear.weight
Parameter containing:
tensor([[ 0.3741+0.j,  0.2382+0.j],
        [ 0.5593+0.j, -0.4443+0.j]], dtype=torch.complex128)
>>> linear(torch.ones(3, 2, dtype=torch.cdouble))
tensor([[0.6122+0.j, 0.1150+0.j],
        [0.6122+0.j, 0.1150+0.j],
        [0.6122+0.j, 0.1150+0.j]], dtype=torch.complex128)
```

to_empty(*, *device: Union[str, torch.device]*) → torch.nn.modules.module.T
Moves the parameters and buffers to the specified device without copying storage.

Parameters **device** (`torch.device`) – The desired device of the parameters and buffers in this module.

Returns self

Return type Module

train(*mode: bool = True*) → torch.nn.modules.module.T
Sets the module in training mode.

This has any effect only on certain modules. See documentations of particular modules for details of their behaviors in training/evaluation mode, if they are affected, e.g. `Dropout`, `BatchNorm`, etc.

Parameters **mode** (*bool*) – whether to set training mode (`True`) or evaluation mode (`False`). Default: `True`.

Returns self

Return type Module

training: bool

type(*dst_type: Union[torch.dtype, str]*) → torch.nn.modules.module.T
Casts all parameters and buffers to `dst_type`.

Note: This method modifies the module in-place.

Parameters **dst_type** (*type or string*) – the desired type

Returns self

Return type Module

xpu(*device: Optional[Union[int, torch.device]] = None*) → torch.nn.modules.module.T
Moves all model parameters and buffers to the XPU.

This also makes associated parameters and buffers different objects. So it should be called before constructing optimizer if the module will live on XPU while being optimized.

Note: This method modifies the module in-place.

Parameters **device** (*int, optional*) – if specified, all parameters will be copied to that device

Returns self

Return type Module

zero_grad(*set_to_none: bool = False*) → None

Sets gradients of all model parameters to zero. See similar function under `torch.optim.Optimizer` for more context.

Parameters **set_to_none** (*bool*) – instead of setting to zero, set the grads to None. See `torch.optim.Optimizer.zero_grad()` for details.

class ConditionalGenerator(*model: torch.nn.modules.module.Module*, *n_channels: int*, *n_units_latent: int*, *cond: Optional[torch.Tensor] = None*, *cond_embedding_n_units_hidden: int = 100*, *device: Any = device(type='cpu')*)

Bases: `torch.nn.modules.module.Module`

Wrapper for making existing CNN generator conditional. Useful for Conditional GANs

Parameters

- **model** – nn.Module Core model.
- **n_channels** – int Number of channels in images
- **n_units_latent** – int Noise size for the input
- **cond** – torch.Tensor The reference conditional
- **cond_embedding_n_units_hidden** – int Size of the conditional embedding layer

T_destination

alias of TypeVar('T_destination', bound=`Dict[str, Any]`)

add_module(*name: str*, *module: Optional[torch.nn.modules.module.Module]*) → None

Adds a child module to the current module.

The module can be accessed as an attribute using the given name.

Parameters

- **name** (*str*) – name of the child module. The child module can be accessed from this module using the given name
- **module** (*Module*) – child module to be added to the module.

apply(*fn: Callable[[torch.nn.modules.module.Module], None]*) → torch.nn.modules.module.T

Applies `fn` recursively to every submodule (as returned by `.children()`) as well as self. Typical use includes initializing the parameters of a model (see also nn-init-doc).

Parameters **fn** (`Module` -> None) – function to be applied to each submodule

Returns self

Return type Module

Example:

```
>>> @torch.no_grad()
>>> def init_weights(m):
>>>     print(m)
>>>     if type(m) == nn.Linear:
>>>         m.weight.fill_(1.0)
>>>         print(m.weight)
>>> net = nn.Sequential(nn.Linear(2, 2), nn.Linear(2, 2))
>>> net.apply(init_weights)
Linear(in_features=2, out_features=2, bias=True)
Parameter containing:
tensor([[1., 1.],
        [1., 1.]], requires_grad=True)
Linear(in_features=2, out_features=2, bias=True)
Parameter containing:
tensor([[1., 1.],
        [1., 1.]], requires_grad=True)
Sequential(
  (0): Linear(in_features=2, out_features=2, bias=True)
  (1): Linear(in_features=2, out_features=2, bias=True)
)
```

bfloat16() → torch.nn.modules.module.T
Casts all floating point parameters and buffers to `bfloat16` datatype.

Note: This method modifies the module in-place.

Returns self

Return type Module

buffers(*recurse: bool = True*) → Iterator[torch.Tensor]
Returns an iterator over module buffers.

Parameters **recurse** (*bool*) – if True, then yields buffers of this module and all submodules. Otherwise, yields only buffers that are direct members of this module.

Yields *torch.Tensor* – module buffer

Example:

```
>>> # xdoctest: +SKIP("undefined vars")
>>> for buf in model.buffers():
>>>     print(type(buf), buf.size())
<class 'torch.Tensor'> (20L,)
<class 'torch.Tensor'> (20L, 1L, 5L, 5L)
```

children() → Iterator[torch.nn.modules.module.Module]
Returns an iterator over immediate children modules.

Yields *Module* – a child module

cpu() → torch.nn.modules.module.T
Moves all model parameters and buffers to the CPU.

Note: This method modifies the module in-place.

Returns self

Return type Module

cuda(*device: Optional[Union[int, torch.device]] = None*) → torch.nn.modules.module.T
Moves all model parameters and buffers to the GPU.

This also makes associated parameters and buffers different objects. So it should be called before constructing optimizer if the module will live on GPU while being optimized.

Note: This method modifies the module in-place.

Parameters **device** (*int, optional*) – if specified, all parameters will be copied to that device

Returns self

Return type Module

double() → torch.nn.modules.module.T
Casts all floating point parameters and buffers to `double` datatype.

Note: This method modifies the module in-place.

Returns self

Return type Module

dump_patches: bool = False

eval() → torch.nn.modules.module.T
Sets the module in evaluation mode.

This has any effect only on certain modules. See documentations of particular modules for details of their behaviors in training/evaluation mode, if they are affected, e.g. `Dropout`, `BatchNorm`, etc.

This is equivalent with `self.train(False)`.

See locally-disable-grad-doc for a comparison between *.eval()* and several similar mechanisms that may be confused with it.

Returns self

Return type Module

extra_repr() → str
Set the extra representation of the module

To print customized extra information, you should re-implement this method in your own modules. Both single-line and multi-line strings are acceptable.

float() → torch.nn.modules.module.T
Casts all floating point parameters and buffers to `float` datatype.

Note: This method modifies the module in-place.

Returns self

Return type Module

forward(*noise: torch.Tensor*, *cond: Optional[torch.Tensor] = None*) → torch.Tensor
Defines the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

get_buffer(*target: str*) → torch.Tensor
Returns the buffer given by `target` if it exists, otherwise throws an error.

See the docstring for `get_submodule` for a more detailed explanation of this method's functionality as well as how to correctly specify `target`.

Parameters **target** – The fully-qualified string name of the buffer to look for. (See `get_submodule` for how to specify a fully-qualified string.)

Returns The buffer referenced by `target`

Return type torch.Tensor

Raises **AttributeError** – If the target string references an invalid path or resolves to something that is not a buffer

get_extra_state() → Any
Returns any extra state to include in the module's state_dict. Implement this and a corresponding *set_extra_state()* for your module if you need to store extra state. This function is called when building the module's *state_dict()*.

Note that extra state should be pickleable to ensure working serialization of the state_dict. We only provide provide backwards compatibility guarantees for serializing Tensors; other objects may break backwards compatibility if their serialized pickled form changes.

Returns Any extra state to store in the module's state_dict

Return type object

get_parameter(*target: str*) → torch.nn.parameter.Parameter
Returns the parameter given by `target` if it exists, otherwise throws an error.

See the docstring for `get_submodule` for a more detailed explanation of this method's functionality as well as how to correctly specify `target`.

Parameters **target** – The fully-qualified string name of the Parameter to look for. (See `get_submodule` for how to specify a fully-qualified string.)

Returns The Parameter referenced by `target`

Return type torch.nn.Parameter

Raises **AttributeError** – If the target string references an invalid path or resolves to something that is not an `nn.Parameter`

get_submodule(*target: str*) → torch.nn.modules.module.Module
Returns the submodule given by `target` if it exists, otherwise throws an error.

For example, let's say you have an `nn.Module` `A` that looks like this:

```
A(
    (net_b): Module(
        (net_c): Module(
            (conv): Conv2d(16, 33, kernel_size=(3, 3), stride=(2, 2))
        )
        (linear): Linear(in_features=100, out_features=200, bias=True)
    )
)
```

(The diagram shows an `nn.Module` `A`. `A` has a nested submodule `net_b`, which itself has two submodules `net_c` and `linear`. `net_c` then has a submodule `conv`.)

To check whether or not we have the `linear` submodule, we would call `get_submodule("net_b.linear")`. To check whether we have the `conv` submodule, we would call `get_submodule("net_b.net_c.conv")`.

The runtime of `get_submodule` is bounded by the degree of module nesting in `target`. A query against `named_modules` achieves the same result, but it is O(N) in the number of transitive modules. So, for a simple check to see if some submodule exists, `get_submodule` should always be used.

Parameters `target` – The fully-qualified string name of the submodule to look for. (See above example for how to specify a fully-qualified string.)

Returns The submodule referenced by `target`

Return type torch.nn.Module

Raises `AttributeError` – If the target string references an invalid path or resolves to something that is not an `nn.Module`

half() → torch.nn.modules.module.T
Casts all floating point parameters and buffers to `half` datatype.

Note: This method modifies the module in-place.

Returns self

Return type Module

ipu(*device: Optional[Union[int, torch.device]] = None*) → torch.nn.modules.module.T
Moves all model parameters and buffers to the IPU.

This also makes associated parameters and buffers different objects. So it should be called before constructing optimizer if the module will live on IPU while being optimized.

Note: This method modifies the module in-place.

Parameters `device` (*int, optional*) – if specified, all parameters will be copied to that device

Returns self

Return type Module

load_state_dict(*state_dict: Mapping[str, Any]*, *strict: bool = True*)
Copies parameters and buffers from *state_dict* into this module and its descendants. If `strict` is `True`, then the keys of *state_dict* must exactly match the keys returned by this module's `state_dict()` function.

Parameters

- **state_dict** (*dict*) – a dict containing parameters and persistent buffers.
- **strict** (*bool, optional*) – whether to strictly enforce that the keys in *state_dict* match the keys returned by this module's `state_dict()` function. Default: `True`

Returns

- **missing_keys** is a list of str containing the missing keys
- **unexpected_keys** is a list of str containing the unexpected keys

Return type `NamedTuple` with `missing_keys` and `unexpected_keys` fields

Note: If a parameter or buffer is registered as `None` and its corresponding key exists in *state_dict*, *load_state_dict()* will raise a `RuntimeError`.

---

modules() → Iterator[torch.nn.modules.module.Module]
Returns an iterator over all modules in the network.

Yields *Module* – a module in the network

Note: Duplicate modules are returned only once. In the following example, `l` will be returned only once.

Example:

```
>>> l = nn.Linear(2, 2)
>>> net = nn.Sequential(l, l)
>>> for idx, m in enumerate(net.modules()):
...     print(idx, '->', m)

0 -> Sequential(
  (0): Linear(in_features=2, out_features=2, bias=True)
  (1): Linear(in_features=2, out_features=2, bias=True)
)
1 -> Linear(in_features=2, out_features=2, bias=True)
```

named_buffers(*prefix: str = ''*, *recurse: bool = True*) → Iterator[Tuple[str, torch.Tensor]]
Returns an iterator over module buffers, yielding both the name of the buffer as well as the buffer itself.

Parameters

- **prefix** (*str*) – prefix to prepend to all buffer names.
- **recurse** (*bool*) – if True, then yields buffers of this module and all submodules. Otherwise, yields only buffers that are direct members of this module.

Yields *(str, torch.Tensor)* – Tuple containing the name and buffer

Example:

```
>>> # xdoctest: +SKIP("undefined vars")
>>> for name, buf in self.named_buffers():
>>>    if name in ['running_var']:
>>>        print(buf.size())
```

named_children() → Iterator[Tuple[str, torch.nn.modules.module.Module]]

Returns an iterator over immediate children modules, yielding both the name of the module as well as the module itself.

Yields *(str, Module)* – Tuple containing a name and child module

Example:

```
>>> # xdoctest: +SKIP("undefined vars")
>>> for name, module in model.named_children():
>>>     if name in ['conv4', 'conv5']:
>>>         print(module)
```

named_modules(*memo: Optional[Set[torch.nn.modules.module.Module]] = None*, *prefix: str = ''*, *remove_duplicate: bool = True*)

Returns an iterator over all modules in the network, yielding both the name of the module as well as the module itself.

Parameters

- **memo** – a memo to store the set of modules already added to the result
- **prefix** – a prefix that will be added to the name of the module
- **remove_duplicate** – whether to remove the duplicated module instances in the result or not

Yields *(str, Module)* – Tuple of name and module

Note: Duplicate modules are returned only once. In the following example, `l` will be returned only once.

Example:

```
>>> l = nn.Linear(2, 2)
>>> net = nn.Sequential(l, l)
>>> for idx, m in enumerate(net.named_modules()):
...     print(idx, '->', m)

0 -> ('', Sequential(
  (0): Linear(in_features=2, out_features=2, bias=True)
  (1): Linear(in_features=2, out_features=2, bias=True)
))
1 -> ('0', Linear(in_features=2, out_features=2, bias=True))
```

named_parameters(*prefix: str = ''*, *recurse: bool = True*) → Iterator[Tuple[str, torch.nn.parameter.Parameter]]

Returns an iterator over module parameters, yielding both the name of the parameter as well as the parameter itself.

Parameters

- **prefix** (*str*) – prefix to prepend to all parameter names.

- **recurse** (*bool*) – if True, then yields parameters of this module and all submodules. Otherwise, yields only parameters that are direct members of this module.

Yields *(str, Parameter)* – Tuple containing the name and parameter

Example:

```
>>> # xdoctest: +SKIP("undefined vars")
>>> for name, param in self.named_parameters():
>>>    if name in ['bias']:
>>>        print(param.size())
```

parameters(*recurse: bool = True*) → Iterator[torch.nn.parameter.Parameter]

Returns an iterator over module parameters.

This is typically passed to an optimizer.

Parameters **recurse** (*bool*) – if True, then yields parameters of this module and all submodules. Otherwise, yields only parameters that are direct members of this module.

Yields *Parameter* – module parameter

Example:

```
>>> # xdoctest: +SKIP("undefined vars")
>>> for param in model.parameters():
>>>     print(type(param), param.size())
<class 'torch.Tensor'> (20L,)
<class 'torch.Tensor'> (20L, 1L, 5L, 5L)
```

register_backward_hook(*hook: Callable[[torch.nn.modules.module.Module, Union[Tuple[torch.Tensor, ...], torch.Tensor], Union[Tuple[torch.Tensor, ...], torch.Tensor]], Union[None, torch.Tensor]]*) → torch.utils.hooks.RemovableHandle

Registers a backward hook on the module.

This function is deprecated in favor of `register_full_backward_hook()` and the behavior of this function will change in future versions.

Returns a handle that can be used to remove the added hook by calling `handle.remove()`

Return type `torch.utils.hooks.RemovableHandle`

register_buffer(*name: str*, *tensor: Optional[torch.Tensor]*, *persistent: bool = True*) → None

Adds a buffer to the module.

This is typically used to register a buffer that should not to be considered a model parameter. For example, BatchNorm's `running_mean` is not a parameter, but is part of the module's state. Buffers, by default, are persistent and will be saved alongside parameters. This behavior can be changed by setting `persistent` to `False`. The only difference between a persistent buffer and a non-persistent buffer is that the latter will not be a part of this module's *state_dict*.

Buffers can be accessed as attributes using given names.

Parameters

- **name** (*str*) – name of the buffer. The buffer can be accessed from this module using the given name
- **tensor** (*Tensor or None*) – buffer to be registered. If `None`, then operations that run on buffers, such as *cuda*, are ignored. If `None`, the buffer is **not** included in the module's *state_dict*.

- **persistent** (*bool*) – whether the buffer is part of this module's *state_dict*.

Example:

```
>>> # xdoctest: +SKIP("undefined vars")
>>> self.register_buffer('running_mean', torch.zeros(num_features))
```

register_forward_hook(*hook: Callable[[...], None]*) → torch.utils.hooks.RemovableHandle

Registers a forward hook on the module.

The hook will be called every time after *forward()* has computed an output. It should have the following signature:

```
hook(module, input, output) -> None or modified output
```

The input contains only the positional arguments given to the module. Keyword arguments won't be passed to the hooks and only to the `forward`. The hook can modify the output. It can modify the input inplace but it will not have effect on forward since this is called after *forward()* is called.

Returns a handle that can be used to remove the added hook by calling `handle.remove()`

Return type `torch.utils.hooks.RemovableHandle`

register_forward_pre_hook(*hook: Callable[[...], None]*) → torch.utils.hooks.RemovableHandle

Registers a forward pre-hook on the module.

The hook will be called every time before *forward()* is invoked. It should have the following signature:

```
hook(module, input) -> None or modified input
```

The input contains only the positional arguments given to the module. Keyword arguments won't be passed to the hooks and only to the `forward`. The hook can modify the input. User can either return a tuple or a single modified value in the hook. We will wrap the value into a tuple if a single value is returned(unless that value is already a tuple).

Returns a handle that can be used to remove the added hook by calling `handle.remove()`

Return type `torch.utils.hooks.RemovableHandle`

register_full_backward_hook(*hook: Callable[[torch.nn.modules.module.Module, Union[Tuple[torch.Tensor, ...], torch.Tensor], Union[Tuple[torch.Tensor, ...], torch.Tensor]], Union[None, torch.Tensor]]*) → torch.utils.hooks.RemovableHandle

Registers a backward hook on the module.

The hook will be called every time the gradients with respect to a module are computed, i.e. the hook will execute if and only if the gradients with respect to module outputs are computed. The hook should have the following signature:

```
hook(module, grad_input, grad_output) -> tuple(Tensor) or None
```

The `grad_input` and `grad_output` are tuples that contain the gradients with respect to the inputs and outputs respectively. The hook should not modify its arguments, but it can optionally return a new gradient with respect to the input that will be used in place of `grad_input` in subsequent computations. `grad_input` will only correspond to the inputs given as positional arguments and all kwarg arguments are ignored. Entries in `grad_input` and `grad_output` will be `None` for all non-Tensor arguments.

For technical reasons, when this hook is applied to a Module, its forward function will receive a view of each Tensor passed to the Module. Similarly the caller will receive a view of each Tensor returned by the Module's forward function.

> **Warning:** Modifying inputs or outputs inplace is not allowed when using backward hooks and will raise an error.

Returns a handle that can be used to remove the added hook by calling `handle.remove()`

Return type `torch.utils.hooks.RemovableHandle`

register_load_state_dict_post_hook(*hook*)

Registers a post hook to be run after module's `load_state_dict` is called.

It should have the following signature:: hook(module, incompatible_keys) -> None

The `module` argument is the current module that this hook is registered on, and the `incompatible_keys` argument is a `NamedTuple` consisting of attributes `missing_keys` and `unexpected_keys`. `missing_keys` is a `list` of `str` containing the missing keys and `unexpected_keys` is a `list` of `str` containing the unexpected keys.

The given incompatible_keys can be modified inplace if needed.

Note that the checks performed when calling *load_state_dict()* with `strict=True` are affected by modifications the hook makes to `missing_keys` or `unexpected_keys`, as expected. Additions to either set of keys will result in an error being thrown when `strict=True`, and clearning out both missing and unexpected keys will avoid an error.

Returns a handle that can be used to remove the added hook by calling `handle.remove()`

Return type `torch.utils.hooks.RemovableHandle`

register_module(*name: str*, *module: Optional[torch.nn.modules.module.Module]*) → None

Alias for *add_module()*.

register_parameter(*name: str*, *param: Optional[torch.nn.parameter.Parameter]*) → None

Adds a parameter to the module.

The parameter can be accessed as an attribute using given name.

Parameters

- **name** (*str*) – name of the parameter. The parameter can be accessed from this module using the given name
- **param** (*Parameter or None*) – parameter to be added to the module. If `None`, then operations that run on parameters, such as *cuda*, are ignored. If `None`, the parameter is **not** included in the module's *state_dict*.

requires_grad_(*requires_grad: bool = True*) → torch.nn.modules.module.T

Change if autograd should record operations on parameters in this module.

This method sets the parameters' `requires_grad` attributes in-place.

This method is helpful for freezing part of the module for finetuning or training parts of a model individually (e.g., GAN training).

See locally-disable-grad-doc for a comparison between *.requires_grad_()* and several similar mechanisms that may be confused with it.

Parameters **requires_grad** (*bool*) – whether autograd should record operations on parameters in this module. Default: `True`.

Returns self

Return type Module

set_extra_state(*state: Any*)
This function is called from *load_state_dict()* to handle any extra state found within the *state_dict*. Implement this function and a corresponding *get_extra_state()* for your module if you need to store extra state within its *state_dict*.

Parameters **state** (*dict*) – Extra state from the *state_dict*

share_memory() → torch.nn.modules.module.T
See `torch.Tensor.share_memory_()`

state_dict(**args*, *destination=None*, *prefix=''*, *keep_vars=False*)
Returns a dictionary containing references to the whole state of the module.

Both parameters and persistent buffers (e.g. running averages) are included. Keys are corresponding parameter and buffer names. Parameters and buffers set to `None` are not included.

Note: The returned object is a shallow copy. It contains references to the module's parameters and buffers.

Warning: Currently `state_dict()` also accepts positional arguments for `destination`, `prefix` and `keep_vars` in order. However, this is being deprecated and keyword arguments will be enforced in future releases.

Warning: Please avoid the use of argument `destination` as it is not designed for end-users.

Parameters

- **destination** (*dict, optional*) – If provided, the state of module will be updated into the dict and the same object is returned. Otherwise, an `OrderedDict` will be created and returned. Default: `None`.
- **prefix** (*str, optional*) – a prefix added to parameter and buffer names to compose the keys in state_dict. Default: `''`.
- **keep_vars** (*bool, optional*) – by default the `Tensor` s returned in the state dict are detached from autograd. If it's set to `True`, detaching will not be performed. Default: `False`.

Returns a dictionary containing a whole state of the module

Return type dict

Example:

```
>>> # xdoctest: +SKIP("undefined vars")
>>> module.state_dict().keys()
['bias', 'weight']
```

to(**args*, ***kwargs*)
Moves and/or casts the parameters and buffers.

This can be called as

to(*device=None*, *dtype=None*, *non_blocking=False*)

to(*dtype*, *non_blocking=False*)

to(*tensor*, *non_blocking=False*)

to(*memory_format=torch.channels_last*)

Its signature is similar to `torch.Tensor.to()`, but only accepts floating point or complex `dtypes`. In addition, this method will only cast the floating point or complex parameters and buffers to `dtype` (if given). The integral parameters and buffers will be moved `device`, if that is given, but with dtypes unchanged. When `non_blocking` is set, it tries to convert/move asynchronously with respect to the host if possible, e.g., moving CPU Tensors with pinned memory to CUDA devices.

See below for examples.

Note: This method modifies the module in-place.

Parameters

- **device** (`torch.device`) – the desired device of the parameters and buffers in this module
- **dtype** (`torch.dtype`) – the desired floating point or complex dtype of the parameters and buffers in this module
- **tensor** (*torch.Tensor*) – Tensor whose dtype and device are the desired dtype and device for all parameters and buffers in this module
- **memory_format** (`torch.memory_format`) – the desired memory format for 4D parameters and buffers in this module (keyword only argument)

Returns self

Return type Module

Examples:

```
>>> # xdoctest: +IGNORE_WANT("non-deterministic")
>>> linear = nn.Linear(2, 2)
>>> linear.weight
Parameter containing:
tensor([[ 0.1913, -0.3420],
        [-0.5113, -0.2325]])
>>> linear.to(torch.double)
Linear(in_features=2, out_features=2, bias=True)
>>> linear.weight
Parameter containing:
tensor([[ 0.1913, -0.3420],
        [-0.5113, -0.2325]], dtype=torch.float64)
>>> # xdoctest: +REQUIRES(env:TORCH_DOCTEST_CUDA1)
>>> gpu1 = torch.device("cuda:1")
>>> linear.to(gpu1, dtype=torch.half, non_blocking=True)
Linear(in_features=2, out_features=2, bias=True)
>>> linear.weight
Parameter containing:
tensor([[ 0.1914, -0.3420],
        [-0.5112, -0.2324]], dtype=torch.float16, device='cuda:1')
>>> cpu = torch.device("cpu")
>>> linear.to(cpu)
```

(continues on next page)

(continued from previous page)

```
Linear(in_features=2, out_features=2, bias=True)
>>> linear.weight
Parameter containing:
tensor([[ 0.1914, -0.3420],
        [-0.5112, -0.2324]], dtype=torch.float16)

>>> linear = nn.Linear(2, 2, bias=None).to(torch.cdouble)
>>> linear.weight
Parameter containing:
tensor([[ 0.3741+0.j,  0.2382+0.j],
        [ 0.5593+0.j, -0.4443+0.j]], dtype=torch.complex128)
>>> linear(torch.ones(3, 2, dtype=torch.cdouble))
tensor([[0.6122+0.j, 0.1150+0.j],
        [0.6122+0.j, 0.1150+0.j],
        [0.6122+0.j, 0.1150+0.j]], dtype=torch.complex128)
```

to_empty(*, *device: Union[str, torch.device]*) → torch.nn.modules.module.T
Moves the parameters and buffers to the specified device without copying storage.

Parameters **device** (`torch.device`) – The desired device of the parameters and buffers in this module.

Returns self

Return type Module

train(*mode: bool = True*) → torch.nn.modules.module.T
Sets the module in training mode.

This has any effect only on certain modules. See documentations of particular modules for details of their behaviors in training/evaluation mode, if they are affected, e.g. `Dropout`, `BatchNorm`, etc.

Parameters **mode** (*bool*) – whether to set training mode (`True`) or evaluation mode (`False`). Default: `True`.

Returns self

Return type Module

training: bool

type(*dst_type: Union[torch.dtype, str]*) → torch.nn.modules.module.T
Casts all parameters and buffers to `dst_type`.

Note: This method modifies the module in-place.

Parameters **dst_type** (*type or string*) – the desired type

Returns self

Return type Module

xpu(*device: Optional[Union[int, torch.device]] = None*) → torch.nn.modules.module.T
Moves all model parameters and buffers to the XPU.

This also makes associated parameters and buffers different objects. So it should be called before constructing optimizer if the module will live on XPU while being optimized.

Note: This method modifies the module in-place.

Parameters **device** (*int, optional*) – if specified, all parameters will be copied to that device

Returns self

Return type Module

zero_grad(*set_to_none: bool = False*) → None
Sets gradients of all model parameters to zero. See similar function under `torch.optim.Optimizer` for more context.

Parameters **set_to_none** (*bool*) – instead of setting to zero, set the grads to None. See `torch.optim.Optimizer.zero_grad()` for details.

class ConvNet(*task_type: str*, *model: torch.nn.modules.module.Module*, *lr: float = 0.001*, *weight_decay: float = 0.001*, *opt_betas: tuple = (0.9, 0.999)*, *n_iter: int = 1000*, *batch_size: int = 500*, *n_iter_print: int = 100*, *random_state: int = 0*, *patience: int = 10*, *n_iter_min: int = 100*, *clipping_value: int = 1*, *early_stopping: bool = True*, *device: Any = device(type='cpu')*)
Bases: `torch.nn.modules.module.Module`

Wrapper for convolutional nets for classification and regression.

Parameters

- **task_type** (*str*) – classifier or regression
- **model** (*nn.Module*) – classification or regression model implementation
- **lr** (*float*) – learning rate for optimizer.
- **weight_decay** (*float*) – l2 (ridge) penalty for the weights.
- **n_iter** (*int*) – Maximum number of iterations.
- **batch_size** (*int*) – Batch size
- **n_iter_print** (*int*) – Number of iterations after which to print updates and check the validation loss.
- **random_state** (*int*) – random_state used
- **patience** (*int*) – Number of iterations to wait before early stopping after decrease in validation loss
- **n_iter_min** (*int*) – Minimum number of iterations to go through before starting early stopping
- **clipping_value** (*int, default 1*) – Gradients clipping value
- **early_stopping** (*bool*) – Enable/disable early stopping

T_destination
alias of TypeVar('T_destination', bound=`Dict[str, Any]`)

add_module(*name: str*, *module: Optional[torch.nn.modules.module.Module]*) → None
Adds a child module to the current module.

The module can be accessed as an attribute using the given name.

Parameters

- **name** (*str*) – name of the child module. The child module can be accessed from this module using the given name
- **module** (*Module*) – child module to be added to the module.

apply(*fn: Callable[[torch.nn.modules.module.Module], None]*) → torch.nn.modules.module.T

Applies `fn` recursively to every submodule (as returned by `.children()`) as well as self. Typical use includes initializing the parameters of a model (see also nn-init-doc).

Parameters **fn** (`Module` -> None) – function to be applied to each submodule

Returns self

Return type Module

Example:

```
>>> @torch.no_grad()
>>> def init_weights(m):
>>>     print(m)
>>>     if type(m) == nn.Linear:
>>>         m.weight.fill_(1.0)
>>>         print(m.weight)
>>> net = nn.Sequential(nn.Linear(2, 2), nn.Linear(2, 2))
>>> net.apply(init_weights)
Linear(in_features=2, out_features=2, bias=True)
Parameter containing:
tensor([[1., 1.],
        [1., 1.]], requires_grad=True)
Linear(in_features=2, out_features=2, bias=True)
Parameter containing:
tensor([[1., 1.],
        [1., 1.]], requires_grad=True)
Sequential(
  (0): Linear(in_features=2, out_features=2, bias=True)
  (1): Linear(in_features=2, out_features=2, bias=True)
)
```

bfloat16() → torch.nn.modules.module.T

Casts all floating point parameters and buffers to `bfloat16` datatype.

Note: This method modifies the module in-place.

Returns self

Return type Module

buffers(*recurse: bool = True*) → Iterator[torch.Tensor]

Returns an iterator over module buffers.

Parameters **recurse** (*bool*) – if True, then yields buffers of this module and all submodules. Otherwise, yields only buffers that are direct members of this module.

Yields *torch.Tensor* – module buffer

Example:

```
>>> # xdoctest: +SKIP("undefined vars")
>>> for buf in model.buffers():
>>>     print(type(buf), buf.size())
<class 'torch.Tensor'> (20L,)
<class 'torch.Tensor'> (20L, 1L, 5L, 5L)
```

children() → Iterator[torch.nn.modules.module.Module]
Returns an iterator over immediate children modules.

Yields *Module* – a child module

cpu() → torch.nn.modules.module.T
Moves all model parameters and buffers to the CPU.

Note: This method modifies the module in-place.

Returns self

Return type Module

cuda(*device: Optional[Union[int, torch.device]] = None*) → torch.nn.modules.module.T
Moves all model parameters and buffers to the GPU.

This also makes associated parameters and buffers different objects. So it should be called before constructing optimizer if the module will live on GPU while being optimized.

Note: This method modifies the module in-place.

Parameters **device** (*int, optional*) – if specified, all parameters will be copied to that device

Returns self

Return type Module

double() → torch.nn.modules.module.T
Casts all floating point parameters and buffers to `double` datatype.

Note: This method modifies the module in-place.

Returns self

Return type Module

dump_patches: bool = False

eval() → torch.nn.modules.module.T
Sets the module in evaluation mode.

This has any effect only on certain modules. See documentations of particular modules for details of their behaviors in training/evaluation mode, if they are affected, e.g. `Dropout`, `BatchNorm`, etc.

This is equivalent with `self.train(False)`.

See locally-disable-grad-doc for a comparison between *.eval()* and several similar mechanisms that may be confused with it.

Returns self

Return type Module

extra_repr() → str
Set the extra representation of the module

To print customized extra information, you should re-implement this method in your own modules. Both single-line and multi-line strings are acceptable.

fit(*X: torch.utils.data.dataset.Dataset*) → *synthcity.plugins.core.models.convnet.ConvNet*

float() → torch.nn.modules.module.T
Casts all floating point parameters and buffers to `float` datatype.

Note: This method modifies the module in-place.

Returns self

Return type Module

forward(*X: torch.Tensor*) → torch.Tensor
Defines the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

---

get_buffer(*target: str*) → torch.Tensor
Returns the buffer given by `target` if it exists, otherwise throws an error.

See the docstring for `get_submodule` for a more detailed explanation of this method's functionality as well as how to correctly specify `target`.

Parameters **target** – The fully-qualified string name of the buffer to look for. (See `get_submodule` for how to specify a fully-qualified string.)

Returns The buffer referenced by `target`

Return type torch.Tensor

Raises **AttributeError** – If the target string references an invalid path or resolves to something that is not a buffer

get_extra_state() → Any
Returns any extra state to include in the module's state_dict. Implement this and a corresponding *set_extra_state()* for your module if you need to store extra state. This function is called when building the module's *state_dict()*.

Note that extra state should be pickleable to ensure working serialization of the state_dict. We only provide provide backwards compatibility guarantees for serializing Tensors; other objects may break backwards compatibility if their serialized pickled form changes.

Returns Any extra state to store in the module's state_dict

Return type object

get_parameter(*target: str*) → torch.nn.parameter.Parameter
Returns the parameter given by `target` if it exists, otherwise throws an error.

See the docstring for `get_submodule` for a more detailed explanation of this method's functionality as well as how to correctly specify `target`.

Parameters **target** – The fully-qualified string name of the Parameter to look for. (See `get_submodule` for how to specify a fully-qualified string.)

Returns The Parameter referenced by `target`

Return type torch.nn.Parameter

Raises **AttributeError** – If the target string references an invalid path or resolves to something that is not an `nn.Parameter`

get_submodule(*target: str*) → torch.nn.modules.module.Module
Returns the submodule given by `target` if it exists, otherwise throws an error.

For example, let's say you have an `nn.Module` `A` that looks like this:

```
A(
    (net_b): Module(
        (net_c): Module(
            (conv): Conv2d(16, 33, kernel_size=(3, 3), stride=(2, 2))
        )
        (linear): Linear(in_features=100, out_features=200, bias=True)
    )
)
```

(The diagram shows an `nn.Module` `A`. `A` has a nested submodule `net_b`, which itself has two submodules `net_c` and `linear`. `net_c` then has a submodule `conv`.)

To check whether or not we have the `linear` submodule, we would call `get_submodule("net_b.linear")`. To check whether we have the `conv` submodule, we would call `get_submodule("net_b.net_c.conv")`.

The runtime of `get_submodule` is bounded by the degree of module nesting in `target`. A query against `named_modules` achieves the same result, but it is O(N) in the number of transitive modules. So, for a simple check to see if some submodule exists, `get_submodule` should always be used.

Parameters **target** – The fully-qualified string name of the submodule to look for. (See above example for how to specify a fully-qualified string.)

Returns The submodule referenced by `target`

Return type torch.nn.Module

Raises **AttributeError** – If the target string references an invalid path or resolves to something that is not an `nn.Module`

half() → torch.nn.modules.module.T
Casts all floating point parameters and buffers to `half` datatype.

Note: This method modifies the module in-place.

Returns self

Return type Module

ipu(*device: Optional[Union[int, torch.device]] = None*) → torch.nn.modules.module.T
Moves all model parameters and buffers to the IPU.

This also makes associated parameters and buffers different objects. So it should be called before constructing optimizer if the module will live on IPU while being optimized.

Note: This method modifies the module in-place.

Parameters **device** (*int, optional*) – if specified, all parameters will be copied to that device

Returns self

Return type Module

load_state_dict(*state_dict: Mapping[str, Any]*, *strict: bool = True*)
Copies parameters and buffers from *state_dict* into this module and its descendants. If `strict` is `True`, then the keys of *state_dict* must exactly match the keys returned by this module's `state_dict()` function.

Parameters

- **state_dict** (*dict*) – a dict containing parameters and persistent buffers.
- **strict** (*bool, optional*) – whether to strictly enforce that the keys in *state_dict* match the keys returned by this module's `state_dict()` function. Default: `True`

Returns

- **missing_keys** is a list of str containing the missing keys
- **unexpected_keys** is a list of str containing the unexpected keys

Return type `NamedTuple` with `missing_keys` and `unexpected_keys` fields

Note: If a parameter or buffer is registered as `None` and its corresponding key exists in *state_dict*, *load_state_dict()* will raise a `RuntimeError`.

---

modules() → Iterator[torch.nn.modules.module.Module]
Returns an iterator over all modules in the network.

Yields *Module* – a module in the network

Note: Duplicate modules are returned only once. In the following example, `l` will be returned only once.

Example:

```
>>> l = nn.Linear(2, 2)
>>> net = nn.Sequential(l, l)
>>> for idx, m in enumerate(net.modules()):
...     print(idx, '->', m)

0 -> Sequential(
  (0): Linear(in_features=2, out_features=2, bias=True)
  (1): Linear(in_features=2, out_features=2, bias=True)
```

(continues on next page)

(continued from previous page)

```
)
1 -> Linear(in_features=2, out_features=2, bias=True)
```

named_buffers(*prefix: str = ''*, *recurse: bool = True*) → Iterator[Tuple[str, torch.Tensor]]

Returns an iterator over module buffers, yielding both the name of the buffer as well as the buffer itself.

Parameters

- **prefix** (*str*) – prefix to prepend to all buffer names.
- **recurse** (*bool*) – if True, then yields buffers of this module and all submodules. Otherwise, yields only buffers that are direct members of this module.

Yields *(str, torch.Tensor)* – Tuple containing the name and buffer

Example:

```
>>> # xdoctest: +SKIP("undefined vars")
>>> for name, buf in self.named_buffers():
>>>    if name in ['running_var']:
>>>        print(buf.size())
```

named_children() → Iterator[Tuple[str, torch.nn.modules.module.Module]]

Returns an iterator over immediate children modules, yielding both the name of the module as well as the module itself.

Yields *(str, Module)* – Tuple containing a name and child module

Example:

```
>>> # xdoctest: +SKIP("undefined vars")
>>> for name, module in model.named_children():
>>>     if name in ['conv4', 'conv5']:
>>>         print(module)
```

named_modules(*memo: Optional[Set[torch.nn.modules.module.Module]] = None*, *prefix: str = ''*, *remove_duplicate: bool = True*)

Returns an iterator over all modules in the network, yielding both the name of the module as well as the module itself.

Parameters

- **memo** – a memo to store the set of modules already added to the result
- **prefix** – a prefix that will be added to the name of the module
- **remove_duplicate** – whether to remove the duplicated module instances in the result or not

Yields *(str, Module)* – Tuple of name and module

Note: Duplicate modules are returned only once. In the following example, `l` will be returned only once.

Example:

```
>>> l = nn.Linear(2, 2)
>>> net = nn.Sequential(l, l)
```

(continues on next page)

(continued from previous page)

```
>>> for idx, m in enumerate(net.named_modules()):
...     print(idx, '->', m)

0 -> ('', Sequential(
  (0): Linear(in_features=2, out_features=2, bias=True)
  (1): Linear(in_features=2, out_features=2, bias=True)
))
1 -> ('0', Linear(in_features=2, out_features=2, bias=True))
```

named_parameters(*prefix: str = ''*, *recurse: bool = True*) → Iterator[Tuple[str, torch.nn.parameter.Parameter]]

Returns an iterator over module parameters, yielding both the name of the parameter as well as the parameter itself.

Parameters

- **prefix** (*str*) – prefix to prepend to all parameter names.
- **recurse** (*bool*) – if True, then yields parameters of this module and all submodules. Otherwise, yields only parameters that are direct members of this module.

Yields *(str, Parameter)* – Tuple containing the name and parameter

Example:

```
>>> # xdoctest: +SKIP("undefined vars")
>>> for name, param in self.named_parameters():
>>>    if name in ['bias']:
>>>        print(param.size())
```

parameters(*recurse: bool = True*) → Iterator[torch.nn.parameter.Parameter]

Returns an iterator over module parameters.

This is typically passed to an optimizer.

Parameters **recurse** (*bool*) – if True, then yields parameters of this module and all submodules. Otherwise, yields only parameters that are direct members of this module.

Yields *Parameter* – module parameter

Example:

```
>>> # xdoctest: +SKIP("undefined vars")
>>> for param in model.parameters():
>>>     print(type(param), param.size())
<class 'torch.Tensor'> (20L,)
<class 'torch.Tensor'> (20L, 1L, 5L, 5L)
```

predict(*X: torch.Tensor*) → torch.Tensor

predict_proba(*X: torch.Tensor*) → torch.Tensor

register_backward_hook(*hook: Callable[[torch.nn.modules.module.Module, Union[Tuple[torch.Tensor, ...], torch.Tensor], Union[Tuple[torch.Tensor, ...], torch.Tensor]], Union[None, torch.Tensor]]*) → torch.utils.hooks.RemovableHandle

Registers a backward hook on the module.

This function is deprecated in favor of `register_full_backward_hook()` and the behavior of this function will change in future versions.

Returns a handle that can be used to remove the added hook by calling `handle.remove()`

Return type `torch.utils.hooks.RemovableHandle`

register_buffer(*name: str*, *tensor: Optional[torch.Tensor]*, *persistent: bool = True*) → None
Adds a buffer to the module.

This is typically used to register a buffer that should not to be considered a model parameter. For example, BatchNorm's `running_mean` is not a parameter, but is part of the module's state. Buffers, by default, are persistent and will be saved alongside parameters. This behavior can be changed by setting `persistent` to `False`. The only difference between a persistent buffer and a non-persistent buffer is that the latter will not be a part of this module's *state_dict*.

Buffers can be accessed as attributes using given names.

Parameters

- **name** (*str*) – name of the buffer. The buffer can be accessed from this module using the given name
- **tensor** (*Tensor or None*) – buffer to be registered. If `None`, then operations that run on buffers, such as *cuda*, are ignored. If `None`, the buffer is **not** included in the module's *state_dict*.
- **persistent** (*bool*) – whether the buffer is part of this module's *state_dict*.

Example:

```
>>> # xdoctest: +SKIP("undefined vars")
>>> self.register_buffer('running_mean', torch.zeros(num_features))
```

register_forward_hook(*hook: Callable[[...], None]*) → torch.utils.hooks.RemovableHandle
Registers a forward hook on the module.

The hook will be called every time after *forward()* has computed an output. It should have the following signature:

```
hook(module, input, output) -> None or modified output
```

The input contains only the positional arguments given to the module. Keyword arguments won't be passed to the hooks and only to the `forward`. The hook can modify the output. It can modify the input inplace but it will not have effect on forward since this is called after *forward()* is called.

Returns a handle that can be used to remove the added hook by calling `handle.remove()`

Return type `torch.utils.hooks.RemovableHandle`

register_forward_pre_hook(*hook: Callable[[...], None]*) → torch.utils.hooks.RemovableHandle
Registers a forward pre-hook on the module.

The hook will be called every time before *forward()* is invoked. It should have the following signature:

```
hook(module, input) -> None or modified input
```

The input contains only the positional arguments given to the module. Keyword arguments won't be passed to the hooks and only to the `forward`. The hook can modify the input. User can either return a tuple or a single modified value in the hook. We will wrap the value into a tuple if a single value is returned(unless that value is already a tuple).

Returns a handle that can be used to remove the added hook by calling `handle.remove()`

Return type `torch.utils.hooks.RemovableHandle`

register_full_backward_hook(*hook: Callable[[torch.nn.modules.module.Module, Union[Tuple[torch.Tensor, ...], torch.Tensor], Union[Tuple[torch.Tensor, ...], torch.Tensor]], Union[None, torch.Tensor]]*) → torch.utils.hooks.RemovableHandle

Registers a backward hook on the module.

The hook will be called every time the gradients with respect to a module are computed, i.e. the hook will execute if and only if the gradients with respect to module outputs are computed. The hook should have the following signature:

```
hook(module, grad_input, grad_output) -> tuple(Tensor) or None
```

The `grad_input` and `grad_output` are tuples that contain the gradients with respect to the inputs and outputs respectively. The hook should not modify its arguments, but it can optionally return a new gradient with respect to the input that will be used in place of `grad_input` in subsequent computations. `grad_input` will only correspond to the inputs given as positional arguments and all kwarg arguments are ignored. Entries in `grad_input` and `grad_output` will be `None` for all non-Tensor arguments.

For technical reasons, when this hook is applied to a Module, its forward function will receive a view of each Tensor passed to the Module. Similarly the caller will receive a view of each Tensor returned by the Module's forward function.

Warning: Modifying inputs or outputs inplace is not allowed when using backward hooks and will raise an error.

Returns a handle that can be used to remove the added hook by calling `handle.remove()`

Return type `torch.utils.hooks.RemovableHandle`

register_load_state_dict_post_hook(*hook*)

Registers a post hook to be run after module's `load_state_dict` is called.

It should have the following signature:: hook(module, incompatible_keys) -> None

The `module` argument is the current module that this hook is registered on, and the `incompatible_keys` argument is a `NamedTuple` consisting of attributes `missing_keys` and `unexpected_keys`. `missing_keys` is a `list` of `str` containing the missing keys and `unexpected_keys` is a `list` of `str` containing the unexpected keys.

The given incompatible_keys can be modified inplace if needed.

Note that the checks performed when calling *load_state_dict()* with `strict=True` are affected by modifications the hook makes to `missing_keys` or `unexpected_keys`, as expected. Additions to either set of keys will result in an error being thrown when `strict=True`, and clearning out both missing and unexpected keys will avoid an error.

Returns a handle that can be used to remove the added hook by calling `handle.remove()`

Return type `torch.utils.hooks.RemovableHandle`

register_module(*name: str*, *module: Optional[torch.nn.modules.module.Module]*) → None

Alias for *add_module()*.

register_parameter(*name: str*, *param: Optional[torch.nn.parameter.Parameter]*) → None

Adds a parameter to the module.

The parameter can be accessed as an attribute using given name.

Parameters

- **name** (*str*) – name of the parameter. The parameter can be accessed from this module using the given name
- **param** (*Parameter or None*) – parameter to be added to the module. If `None`, then operations that run on parameters, such as *cuda*, are ignored. If `None`, the parameter is **not** included in the module's *state_dict*.

requires_grad_(*requires_grad: bool = True*) → torch.nn.modules.module.T
Change if autograd should record operations on parameters in this module.

This method sets the parameters' `requires_grad` attributes in-place.

This method is helpful for freezing part of the module for finetuning or training parts of a model individually (e.g., GAN training).

See locally-disable-grad-doc for a comparison between *.requires_grad_()* and several similar mechanisms that may be confused with it.

Parameters **requires_grad** (*bool*) – whether autograd should record operations on parameters in this module. Default: `True`.

Returns self

Return type Module

score(*X: torch.Tensor*, *y: torch.Tensor*) → *float*

set_extra_state(*state: Any*)
This function is called from *load_state_dict()* to handle any extra state found within the *state_dict*. Implement this function and a corresponding *get_extra_state()* for your module if you need to store extra state within its *state_dict*.

Parameters **state** (*dict*) – Extra state from the *state_dict*

share_memory() → torch.nn.modules.module.T
See `torch.Tensor.share_memory_()`

state_dict(**args*, *destination=None*, *prefix=''*, *keep_vars=False*)
Returns a dictionary containing references to the whole state of the module.

Both parameters and persistent buffers (e.g. running averages) are included. Keys are corresponding parameter and buffer names. Parameters and buffers set to `None` are not included.

Note: The returned object is a shallow copy. It contains references to the module's parameters and buffers.

Warning: Currently `state_dict()` also accepts positional arguments for `destination`, `prefix` and `keep_vars` in order. However, this is being deprecated and keyword arguments will be enforced in future releases.

Warning: Please avoid the use of argument `destination` as it is not designed for end-users.

Parameters

- **destination** (*dict, optional*) – If provided, the state of module will be updated into the dict and the same object is returned. Otherwise, an `OrderedDict` will be created and returned. Default: `None`.

- **prefix** (*str, optional*) – a prefix added to parameter and buffer names to compose the keys in state_dict. Default: ''.
- **keep_vars** (*bool, optional*) – by default the Tensor s returned in the state dict are detached from autograd. If it's set to True, detaching will not be performed. Default: False.

Returns a dictionary containing a whole state of the module

Return type dict

Example:

```
>>> # xdoctest: +SKIP("undefined vars")
>>> module.state_dict().keys()
['bias', 'weight']
```

to(**args*, ***kwargs*)

Moves and/or casts the parameters and buffers.

This can be called as

to(*device=None*, *dtype=None*, *non_blocking=False*)

to(*dtype*, *non_blocking=False*)

to(*tensor*, *non_blocking=False*)

to(*memory_format=torch.channels_last*)

Its signature is similar to torch.Tensor.to(), but only accepts floating point or complex dtypes. In addition, this method will only cast the floating point or complex parameters and buffers to dtype (if given). The integral parameters and buffers will be moved device, if that is given, but with dtypes unchanged. When non_blocking is set, it tries to convert/move asynchronously with respect to the host if possible, e.g., moving CPU Tensors with pinned memory to CUDA devices.

See below for examples.

Note: This method modifies the module in-place.

Parameters

- **device** (torch.device) – the desired device of the parameters and buffers in this module
- **dtype** (torch.dtype) – the desired floating point or complex dtype of the parameters and buffers in this module
- **tensor** (*torch.Tensor*) – Tensor whose dtype and device are the desired dtype and device for all parameters and buffers in this module
- **memory_format** (torch.memory_format) – the desired memory format for 4D parameters and buffers in this module (keyword only argument)

Returns self

Return type Module

Examples:

```
>>> # xdoctest: +IGNORE_WANT("non-deterministic")
>>> linear = nn.Linear(2, 2)
>>> linear.weight
Parameter containing:
tensor([[ 0.1913, -0.3420],
        [-0.5113, -0.2325]])
>>> linear.to(torch.double)
Linear(in_features=2, out_features=2, bias=True)
>>> linear.weight
Parameter containing:
tensor([[ 0.1913, -0.3420],
        [-0.5113, -0.2325]], dtype=torch.float64)
>>> # xdoctest: +REQUIRES(env:TORCH_DOCTEST_CUDA1)
>>> gpu1 = torch.device("cuda:1")
>>> linear.to(gpu1, dtype=torch.half, non_blocking=True)
Linear(in_features=2, out_features=2, bias=True)
>>> linear.weight
Parameter containing:
tensor([[ 0.1914, -0.3420],
        [-0.5112, -0.2324]], dtype=torch.float16, device='cuda:1')
>>> cpu = torch.device("cpu")
>>> linear.to(cpu)
Linear(in_features=2, out_features=2, bias=True)
>>> linear.weight
Parameter containing:
tensor([[ 0.1914, -0.3420],
        [-0.5112, -0.2324]], dtype=torch.float16)

>>> linear = nn.Linear(2, 2, bias=None).to(torch.cdouble)
>>> linear.weight
Parameter containing:
tensor([[ 0.3741+0.j,  0.2382+0.j],
        [ 0.5593+0.j, -0.4443+0.j]], dtype=torch.complex128)
>>> linear(torch.ones(3, 2, dtype=torch.cdouble))
tensor([[0.6122+0.j, 0.1150+0.j],
        [0.6122+0.j, 0.1150+0.j],
        [0.6122+0.j, 0.1150+0.j]], dtype=torch.complex128)
```

to_empty(*, *device: Union[str, torch.device]*) → torch.nn.modules.module.T
Moves the parameters and buffers to the specified device without copying storage.

Parameters **device** (torch.device) – The desired device of the parameters and buffers in this module.

Returns self

Return type Module

train(*mode: bool = True*) → torch.nn.modules.module.T
Sets the module in training mode.

This has any effect only on certain modules. See documentations of particular modules for details of their behaviors in training/evaluation mode, if they are affected, e.g. Dropout, BatchNorm, etc.

Parameters **mode** (*bool*) – whether to set training mode (True) or evaluation mode (False). Default: True.

Returns self

Return type Module

`training: bool`

`type`(*dst_type: Union[torch.dtype, str]*) → torch.nn.modules.module.T
Casts all parameters and buffers to `dst_type`.

Note: This method modifies the module in-place.

Parameters `dst_type` (*type or string*) – the desired type

Returns self

Return type Module

`xpu`(*device: Optional[Union[int, torch.device]] = None*) → torch.nn.modules.module.T
Moves all model parameters and buffers to the XPU.

This also makes associated parameters and buffers different objects. So it should be called before constructing optimizer if the module will live on XPU while being optimized.

Note: This method modifies the module in-place.

Parameters `device` (*int, optional*) – if specified, all parameters will be copied to that device

Returns self

Return type Module

`zero_grad`(*set_to_none: bool = False*) → None
Sets gradients of all model parameters to zero. See similar function under `torch.optim.Optimizer` for more context.

Parameters `set_to_none` (*bool*) – instead of setting to zero, set the grads to None. See `torch.optim.Optimizer.zero_grad()` for details.

`map_nonlin`(*nonlin: str*) → <monai.networks.layers.factories.LayerFactory object at 0x7f663d589f40>

`suggest_image_classifier_arch`(*n_channels: int*, *height: int*, *width: int*, *classes: int*, *n_residual_units: int = 2*, *nonlin: str = 'prelu'*, *dropout: float = 0.2*, *last_nonlin: str = 'softmax'*, *device: Any = device(type='cpu')*, *strategy: str = 'predefined'*, *lr: float = 0.001*, *weight_decay: float = 0.001*, *opt_betas: tuple = (0.9, 0.999)*, *n_iter: int = 1000*, *batch_size: int = 500*, *n_iter_print: int = 100*, *random_state: int = 0*, *patience: int = 10*, *n_iter_min: int = 100*, *clipping_value: int = 1*, *early_stopping: bool = True*) → *synthcity.plugins.core.models.convnet.ConvNet*
Helper for selecting compatible architecture for image classifiers.

Parameters

- `n_channels` – int Number of channels in the image
- `height` – int Image height
- `width` – int Image width

- **classes** – int Number of output classes
- **nonlin** – str name of the activation activation layers. Can be relu, elu, prelu or leaky_relu
- **last_act** – str output activation
- **dropout** – float = 0.2 Dropout value
- **n_residual_units** – int integer stating number of convolutions in residual units, 0 means no residual units
- **device** –

  str PyTorch device. cpu, cuda

  # Training

  lr: float learning rate for optimizer.

  weight_decay: float l2 (ridge) penalty for the weights.

  n_iter: int Maximum number of iterations.

  batch_size: int Batch size

  n_iter_print: int Number of iterations after which to print updates and check the validation loss.

  random_state: int random_state used

  patience: int Number of iterations to wait before early stopping after decrease in validation loss

  n_iter_min: int Minimum number of iterations to go through before starting early stopping

  clipping_value: int, default 1 Gradients clipping value

  early_stopping: bool Enable/disable early stopping

suggest_image_generator_discriminator_arch(*n_units_latent: int*, *n_channels: int*, *height: int*, *width: int*, *generator_dropout: float = 0.2*, *generator_nonlin: str = 'prelu'*, *generator_n_residual_units: int = 2*, *discriminator_dropout: float = 0.2*, *discriminator_nonlin: str = 'prelu'*, *discriminator_n_residual_units: int = 2*, *device: Any = device(type='cpu')*, *strategy: str = 'predefined'*, *cond: Optional[torch.Tensor] = None*, *cond_embedding_n_units_hidden: int = 100*) → Tuple[*synthcity.plugins.core.models.convnet.ConditionalGenerator*, *synthcity.plugins.core.models.convnet.ConditionalDiscriminator*]

Helper for selecting compatible architecture for image generators and discriminators.

Parameters

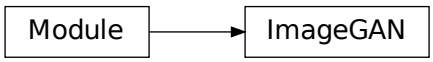
- **n_units_latent** – int, Input size for the generator
- **n_channels** – int Number of channels in the image
- **height** – int Image height
- **width** – int Image width
- **generator_dropout** – float = 0.2 Dropout value for the generator

- **generator_nonlin** – str name of the activation activation layers in the generator. Can be relu, elu, prelu or leaky_relu
- **generator_n_residual_units** – int integer stating number of convolutions in residual units for the generator, 0 means no residual units
- **discriminator_dropout** – float = 0.2 Dropout value for the discriminator
- **discriminator_nonlin** – str name of the activation activation layers in the discriminator. Can be relu, elu, prelu or leaky_relu
- **discriminator_n_residual_units** – int integer stating number of convolutions in residual units for the discriminator, 0 means no residual units
- **device** – str PyTorch device. cpu, cuda
- **strategy** – str Which suggestion to use. Options:
  - predefined: a few hardcoded architectures for certain image shapes.
  - ...

synthcity.plugins.core.models.image_gan module

class ImageGAN(*image_generator: torch.nn.modules.module.Module*, *image_discriminator: torch.nn.modules.module.Module*, *n_units_latent: int*, *n_channels: int*, *generator_n_iter: int = 500*, *generator_lr: float = 0.0002*, *generator_weight_decay: float = 0.001*, *generator_opt_betas: tuple = (0.9, 0.999)*, *generator_extra_penalties: list = []*, *generator_extra_penalty_cbks: List[Callable] = []*, *discriminator_n_iter: int = 1*, *discriminator_lr: float = 0.0002*, *discriminator_weight_decay: float = 0.001*, *discriminator_opt_betas: tuple = (0.9, 0.999)*, *batch_size: int = 100*, *random_state: int = 0*, *clipping_value: int = 1*, *lambda_gradient_penalty: float = 10*, *lambda_identifiability_penalty: float = 0.1*, *device: Any = device(type='cpu')*, *n_iter_min: int = 100*, *n_iter_print: int = 1*, *plot_progress: int = False*, *patience: int = 20*, *patience_metric: Optional[synthcity.metrics.weighted_metrics.WeightedMetrics] = None*, *dataloader_sampler: Optional[torch.utils.data.sampler.Sampler] = None*, *dp_enabled: bool = False*, *dp_delta: Optional[float] = None*, *dp_epsilon: float = 3*, *dp_max_grad_norm: float = 2*, *dp_secure_mode: bool = False*)

Bases: `torch.nn.modules.module.Module`



Basic GAN implementation.

Parameters

- **image_generator** – nn.Module Generator model
- **image_discriminator** – nn.Module Discriminator model
- **n_units_latent** – int Number of latent units
- **n_channels** – int Number of channels in the image
- **generator_n_iter** – int Maximum number of iterations in the Generator.

- **generator_lr** – float = 2e-4 Generator learning rate, used by the Adam optimizer
- **generator_weight_decay** – float = 1e-3 Generator weight decay, used by the Adam optimizer
- **generator_opt_betas** – tuple = (0.9, 0.999) Generator initial decay rates, used by the Adam Optimizer
- **generator_extra_penalties** – list Additional penalties for the generator. Values: "identifiability_penalty"
- **generator_extra_penalty_cbks** – List[Callable] Additional loss callabacks for the generator. Used by the TabularGAN for the conditional loss
- **discriminator_n_iter** – int Maximum number of iterations in the discriminator.
- **discriminator_lr** – float Discriminator learning rate, used by the Adam optimizer
- **discriminator_weight_decay** – float Discriminator weight decay, used by the Adam optimizer
- **discriminator_opt_betas** – tuple Initial weight decays for the Adam optimizer
- **batch_size** – int Batch size
- **random_state** – int random_state used
- **clipping_value** – int, default 0 Gradients clipping value. Zero disables the feature
- **lambda_gradient_penalty** – float = 10 Weight for the gradient penalty
- **lambda_identifiability_penalty** – float = 0.1 Weight for the identifiability penalty, if enabled
- **dataloader_sampler** – Optional[sampler.Sampler] Optional sampler for the dataloader, useful for conditional sampling
- **device** – Any = DEVICE CUDA/CPU
- **stopping** (*# early*) –
- **n_iter_print** – int Number of iterations after which to print updates and check the validation loss.
- **n_iter_min** – int Minimum number of iterations to go through before starting early stopping
- **patience** – int Max number of iterations without any improvement before early stopping is trigged.
- **patience_metric** – Optional[WeightedMetrics] If not None, the metric is used for evaluation the criterion for early stopping.
- **settings** (*# privacy*) –
- **dp_enabled** – bool Train the discriminator with Differential Privacy guarantees
- **dp_delta** – Optional[float] Optional DP delta: the probability of information accidentally being leaked. Usually 1 / len(dataset)
- **dp_epsilon** – float = 3 DP epsilon: privacy budget, which is a measure of the amount of privacy that is preserved by a given algorithm. Epsilon is a number that represents the maximum amount of information that an adversary can learn about an individual from the output of a differentially private algorithm. The smaller the value of epsilon, the more private the algorithm is. For example, an algorithm with an epsilon of 0.1 preserves more privacy than an algorithm with an epsilon of 1.0.

- **dp_max_grad_norm** – float max grad norm used for gradient clipping
- **dp_secure_mode** – bool = False, if True uses noise generation approach robust to floating point arithmetic attacks.

T_destination
alias of TypeVar('T_destination', bound=Dict[str, Any])

add_module(*name: str*, *module: Optional[torch.nn.modules.module.Module]*) → None
Adds a child module to the current module.

The module can be accessed as an attribute using the given name.

Parameters

- **name** (*str*) – name of the child module. The child module can be accessed from this module using the given name
- **module** (*Module*) – child module to be added to the module.

apply(*fn: Callable[[torch.nn.modules.module.Module], None]*) → torch.nn.modules.module.T
Applies `fn` recursively to every submodule (as returned by `.children()`) as well as self. Typical use includes initializing the parameters of a model (see also nn-init-doc).

Parameters **fn** (Module -> None) – function to be applied to each submodule

Returns self

Return type Module

Example:

```
>>> @torch.no_grad()
>>> def init_weights(m):
>>>     print(m)
>>>     if type(m) == nn.Linear:
>>>         m.weight.fill_(1.0)
>>>         print(m.weight)
>>> net = nn.Sequential(nn.Linear(2, 2), nn.Linear(2, 2))
>>> net.apply(init_weights)
Linear(in_features=2, out_features=2, bias=True)
Parameter containing:
tensor([[1., 1.],
        [1., 1.]], requires_grad=True)
Linear(in_features=2, out_features=2, bias=True)
Parameter containing:
tensor([[1., 1.],
        [1., 1.]], requires_grad=True)
Sequential(
  (0): Linear(in_features=2, out_features=2, bias=True)
  (1): Linear(in_features=2, out_features=2, bias=True)
)
```

bfloat16() → torch.nn.modules.module.T
Casts all floating point parameters and buffers to `bfloat16` datatype.

Note: This method modifies the module in-place.

Returns self

Return type Module

buffers(*recurse: bool = True*) → Iterator[torch.Tensor]
Returns an iterator over module buffers.

Parameters **recurse** (*bool*) – if True, then yields buffers of this module and all submodules. Otherwise, yields only buffers that are direct members of this module.

Yields *torch.Tensor* – module buffer

Example:

```
>>> # xdoctest: +SKIP("undefined vars")
>>> for buf in model.buffers():
>>>     print(type(buf), buf.size())
<class 'torch.Tensor'> (20L,)
<class 'torch.Tensor'> (20L, 1L, 5L, 5L)
```

children() → Iterator[torch.nn.modules.module.Module]
Returns an iterator over immediate children modules.

Yields *Module* – a child module

cpu() → torch.nn.modules.module.T
Moves all model parameters and buffers to the CPU.

Note: This method modifies the module in-place.

Returns self

Return type Module

cuda(*device: Optional[Union[int, torch.device]] = None*) → torch.nn.modules.module.T
Moves all model parameters and buffers to the GPU.

This also makes associated parameters and buffers different objects. So it should be called before constructing optimizer if the module will live on GPU while being optimized.

Note: This method modifies the module in-place.

Parameters **device** (*int, optional*) – if specified, all parameters will be copied to that device

Returns self

Return type Module

dataloader(*dataset: torch.utils.data.dataset.Dataset*, *cond: Optional[torch.Tensor] = None*) → torch.utils.data.dataloader.DataLoader

double() → torch.nn.modules.module.T
Casts all floating point parameters and buffers to `double` datatype.

Note: This method modifies the module in-place.

Returns self

Return type Module

dump_patches: bool = False

eval() → torch.nn.modules.module.T
Sets the module in evaluation mode.

This has any effect only on certain modules. See documentations of particular modules for details of their behaviors in training/evaluation mode, if they are affected, e.g. `Dropout`, `BatchNorm`, etc.

This is equivalent with `self.train(False)`.

See locally-disable-grad-doc for a comparison between *.eval()* and several similar mechanisms that may be confused with it.

Returns self

Return type Module

extra_repr() → str
Set the extra representation of the module

To print customized extra information, you should re-implement this method in your own modules. Both single-line and multi-line strings are acceptable.

fit(*X:* synthcity.plugins.core.dataset.FlexibleDataset, *cond: Optional[torch.Tensor] = None*, *fake_labels_generator: Optional[Callable] = None*, *true_labels_generator: Optional[Callable] = None*) → *synthcity.plugins.core.models.image_gan.ImageGAN*

float() → torch.nn.modules.module.T
Casts all floating point parameters and buffers to `float` datatype.

Note: This method modifies the module in-place.

Returns self

Return type Module

forward(*count: int*, *cond: Optional[torch.Tensor] = None*) → torch.Tensor
Defines the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

---

generate(*count: int*, *cond: Optional[torch.Tensor] = None*) → torch.Tensor

get_buffer(*target: str*) → torch.Tensor
Returns the buffer given by `target` if it exists, otherwise throws an error.

See the docstring for `get_submodule` for a more detailed explanation of this method's functionality as well as how to correctly specify `target`.

Parameters **target** – The fully-qualified string name of the buffer to look for. (See `get_submodule` for how to specify a fully-qualified string.)

Returns The buffer referenced by `target`

Return type torch.Tensor

Raises **AttributeError** – If the target string references an invalid path or resolves to something that is not a buffer

get_extra_state() → Any

Returns any extra state to include in the module's state_dict. Implement this and a corresponding *set_extra_state()* for your module if you need to store extra state. This function is called when building the module's *state_dict()*.

Note that extra state should be pickleable to ensure working serialization of the state_dict. We only provide provide backwards compatibility guarantees for serializing Tensors; other objects may break backwards compatibility if their serialized pickled form changes.

Returns Any extra state to store in the module's state_dict

Return type object

get_parameter(*target: str*) → torch.nn.parameter.Parameter

Returns the parameter given by `target` if it exists, otherwise throws an error.

See the docstring for `get_submodule` for a more detailed explanation of this method's functionality as well as how to correctly specify `target`.

Parameters **target** – The fully-qualified string name of the Parameter to look for. (See `get_submodule` for how to specify a fully-qualified string.)

Returns The Parameter referenced by `target`

Return type torch.nn.Parameter

Raises **AttributeError** – If the target string references an invalid path or resolves to something that is not an `nn.Parameter`

get_submodule(*target: str*) → torch.nn.modules.module.Module

Returns the submodule given by `target` if it exists, otherwise throws an error.

For example, let's say you have an `nn.Module` `A` that looks like this:

```
A(
    (net_b): Module(
        (net_c): Module(
            (conv): Conv2d(16, 33, kernel_size=(3, 3), stride=(2, 2))
        )
        (linear): Linear(in_features=100, out_features=200, bias=True)
    )
)
```

(The diagram shows an `nn.Module` `A`. `A` has a nested submodule `net_b`, which itself has two submodules `net_c` and `linear`. `net_c` then has a submodule `conv`.)

To check whether or not we have the `linear` submodule, we would call `get_submodule("net_b.linear")`. To check whether we have the `conv` submodule, we would call `get_submodule("net_b.net_c.conv")`.

The runtime of `get_submodule` is bounded by the degree of module nesting in `target`. A query against `named_modules` achieves the same result, but it is O(N) in the number of transitive modules. So, for a simple check to see if some submodule exists, `get_submodule` should always be used.

Parameters **target** – The fully-qualified string name of the submodule to look for. (See above example for how to specify a fully-qualified string.)

Returns The submodule referenced by `target`

Return type torch.nn.Module

Raises **AttributeError** – If the target string references an invalid path or resolves to something that is not an `nn.Module`

half() → torch.nn.modules.module.T
Casts all floating point parameters and buffers to `half` datatype.

Note: This method modifies the module in-place.

Returns self

Return type Module

ipu(*device: Optional[Union[int, torch.device]] = None*) → torch.nn.modules.module.T
Moves all model parameters and buffers to the IPU.

This also makes associated parameters and buffers different objects. So it should be called before constructing optimizer if the module will live on IPU while being optimized.

Note: This method modifies the module in-place.

Parameters **device** (*int, optional*) – if specified, all parameters will be copied to that device

Returns self

Return type Module

load_state_dict(*state_dict: Mapping[str, Any]*, *strict: bool = True*)
Copies parameters and buffers from *state_dict* into this module and its descendants. If `strict` is `True`, then the keys of *state_dict* must exactly match the keys returned by this module's `state_dict()` function.

Parameters

- **state_dict** (*dict*) – a dict containing parameters and persistent buffers.
- **strict** (*bool, optional*) – whether to strictly enforce that the keys in *state_dict* match the keys returned by this module's `state_dict()` function. Default: `True`

Returns

- **missing_keys** is a list of str containing the missing keys
- **unexpected_keys** is a list of str containing the unexpected keys

Return type `NamedTuple` with `missing_keys` and `unexpected_keys` fields

Note: If a parameter or buffer is registered as `None` and its corresponding key exists in *state_dict*, *load_state_dict()* will raise a `RuntimeError`.

---

modules() → Iterator[torch.nn.modules.module.Module]
Returns an iterator over all modules in the network.

Yields *Module* – a module in the network

Note: Duplicate modules are returned only once. In the following example, `l` will be returned only once.

Example:

```
>>> l = nn.Linear(2, 2)
>>> net = nn.Sequential(l, l)
>>> for idx, m in enumerate(net.modules()):
...     print(idx, '->', m)

0 -> Sequential(
  (0): Linear(in_features=2, out_features=2, bias=True)
  (1): Linear(in_features=2, out_features=2, bias=True)
)
1 -> Linear(in_features=2, out_features=2, bias=True)
```

named_buffers(*prefix: str = ''*, *recurse: bool = True*) → Iterator[Tuple[str, torch.Tensor]]

Returns an iterator over module buffers, yielding both the name of the buffer as well as the buffer itself.

Parameters

- **prefix** (*str*) – prefix to prepend to all buffer names.
- **recurse** (*bool*) – if True, then yields buffers of this module and all submodules. Otherwise, yields only buffers that are direct members of this module.

Yields *(str, torch.Tensor)* – Tuple containing the name and buffer

Example:

```
>>> # xdoctest: +SKIP("undefined vars")
>>> for name, buf in self.named_buffers():
>>>    if name in ['running_var']:
>>>        print(buf.size())
```

named_children() → Iterator[Tuple[str, torch.nn.modules.module.Module]]

Returns an iterator over immediate children modules, yielding both the name of the module as well as the module itself.

Yields *(str, Module)* – Tuple containing a name and child module

Example:

```
>>> # xdoctest: +SKIP("undefined vars")
>>> for name, module in model.named_children():
>>>     if name in ['conv4', 'conv5']:
>>>         print(module)
```

named_modules(*memo: Optional[Set[torch.nn.modules.module.Module]] = None*, *prefix: str = ''*, *remove_duplicate: bool = True*)

Returns an iterator over all modules in the network, yielding both the name of the module as well as the module itself.

Parameters

- **memo** – a memo to store the set of modules already added to the result
- **prefix** – a prefix that will be added to the name of the module

- **remove_duplicate** – whether to remove the duplicated module instances in the result or not

Yields *(str, Module)* – Tuple of name and module

Note: Duplicate modules are returned only once. In the following example, `l` will be returned only once.

Example:

```
>>> l = nn.Linear(2, 2)
>>> net = nn.Sequential(l, l)
>>> for idx, m in enumerate(net.named_modules()):
...     print(idx, '->', m)

0 -> ('', Sequential(
  (0): Linear(in_features=2, out_features=2, bias=True)
  (1): Linear(in_features=2, out_features=2, bias=True)
))
1 -> ('0', Linear(in_features=2, out_features=2, bias=True))
```

named_parameters(*prefix: str = ''*, *recurse: bool = True*) → Iterator[Tuple[str, torch.nn.parameter.Parameter]]

Returns an iterator over module parameters, yielding both the name of the parameter as well as the parameter itself.

Parameters

- **prefix** (*str*) – prefix to prepend to all parameter names.
- **recurse** (*bool*) – if True, then yields parameters of this module and all submodules. Otherwise, yields only parameters that are direct members of this module.

Yields *(str, Parameter)* – Tuple containing the name and parameter

Example:

```
>>> # xdoctest: +SKIP("undefined vars")
>>> for name, param in self.named_parameters():
>>>    if name in ['bias']:
>>>        print(param.size())
```

parameters(*recurse: bool = True*) → Iterator[torch.nn.parameter.Parameter]

Returns an iterator over module parameters.

This is typically passed to an optimizer.

Parameters **recurse** (*bool*) – if True, then yields parameters of this module and all submodules. Otherwise, yields only parameters that are direct members of this module.

Yields *Parameter* – module parameter

Example:

```
>>> # xdoctest: +SKIP("undefined vars")
>>> for param in model.parameters():
>>>     print(type(param), param.size())
<class 'torch.Tensor'> (20L,)
<class 'torch.Tensor'> (20L, 1L, 5L, 5L)
```

register_backward_hook(*hook: Callable[[torch.nn.modules.module.Module, Union[Tuple[torch.Tensor, ...], torch.Tensor], Union[Tuple[torch.Tensor, ...], torch.Tensor]], Union[None, torch.Tensor]]*) → torch.utils.hooks.RemovableHandle

Registers a backward hook on the module.

This function is deprecated in favor of `register_full_backward_hook()` and the behavior of this function will change in future versions.

Returns a handle that can be used to remove the added hook by calling `handle.remove()`

Return type `torch.utils.hooks.RemovableHandle`

register_buffer(*name: str*, *tensor: Optional[torch.Tensor]*, *persistent: bool = True*) → None

Adds a buffer to the module.

This is typically used to register a buffer that should not to be considered a model parameter. For example, BatchNorm's `running_mean` is not a parameter, but is part of the module's state. Buffers, by default, are persistent and will be saved alongside parameters. This behavior can be changed by setting `persistent` to `False`. The only difference between a persistent buffer and a non-persistent buffer is that the latter will not be a part of this module's *state_dict*.

Buffers can be accessed as attributes using given names.

Parameters

- **name** (*str*) – name of the buffer. The buffer can be accessed from this module using the given name
- **tensor** (*Tensor or None*) – buffer to be registered. If `None`, then operations that run on buffers, such as *cuda*, are ignored. If `None`, the buffer is **not** included in the module's *state_dict*.
- **persistent** (*bool*) – whether the buffer is part of this module's *state_dict*.

Example:

```
>>> # xdoctest: +SKIP("undefined vars")
>>> self.register_buffer('running_mean', torch.zeros(num_features))
```

register_forward_hook(*hook: Callable[[...], None]*) → torch.utils.hooks.RemovableHandle

Registers a forward hook on the module.

The hook will be called every time after *forward()* has computed an output. It should have the following signature:

```
hook(module, input, output) -> None or modified output
```

The input contains only the positional arguments given to the module. Keyword arguments won't be passed to the hooks and only to the `forward`. The hook can modify the output. It can modify the input inplace but it will not have effect on forward since this is called after *forward()* is called.

Returns a handle that can be used to remove the added hook by calling `handle.remove()`

Return type `torch.utils.hooks.RemovableHandle`

register_forward_pre_hook(*hook: Callable[[...], None]*) → torch.utils.hooks.RemovableHandle

Registers a forward pre-hook on the module.

The hook will be called every time before *forward()* is invoked. It should have the following signature:

```
hook(module, input) -> None or modified input
```

The input contains only the positional arguments given to the module. Keyword arguments won't be passed to the hooks and only to the `forward`. The hook can modify the input. User can either return a tuple or a single modified value in the hook. We will wrap the value into a tuple if a single value is returned(unless that value is already a tuple).

Returns a handle that can be used to remove the added hook by calling `handle.remove()`

Return type `torch.utils.hooks.RemovableHandle`

register_full_backward_hook(*hook: Callable[[torch.nn.modules.module.Module, Union[Tuple[torch.Tensor, ...], torch.Tensor], Union[Tuple[torch.Tensor, ...], torch.Tensor]], Union[None, torch.Tensor]]*) → torch.utils.hooks.RemovableHandle

Registers a backward hook on the module.

The hook will be called every time the gradients with respect to a module are computed, i.e. the hook will execute if and only if the gradients with respect to module outputs are computed. The hook should have the following signature:

```
hook(module, grad_input, grad_output) -> tuple(Tensor) or None
```

The `grad_input` and `grad_output` are tuples that contain the gradients with respect to the inputs and outputs respectively. The hook should not modify its arguments, but it can optionally return a new gradient with respect to the input that will be used in place of `grad_input` in subsequent computations. `grad_input` will only correspond to the inputs given as positional arguments and all kwarg arguments are ignored. Entries in `grad_input` and `grad_output` will be `None` for all non-Tensor arguments.

For technical reasons, when this hook is applied to a Module, its forward function will receive a view of each Tensor passed to the Module. Similarly the caller will receive a view of each Tensor returned by the Module's forward function.

Warning: Modifying inputs or outputs inplace is not allowed when using backward hooks and will raise an error.

Returns a handle that can be used to remove the added hook by calling `handle.remove()`

Return type `torch.utils.hooks.RemovableHandle`

register_load_state_dict_post_hook(*hook*)

Registers a post hook to be run after module's `load_state_dict` is called.

It should have the following signature:: hook(module, incompatible_keys) -> None

The `module` argument is the current module that this hook is registered on, and the `incompatible_keys` argument is a `NamedTuple` consisting of attributes `missing_keys` and `unexpected_keys`. `missing_keys` is a `list` of `str` containing the missing keys and `unexpected_keys` is a `list` of `str` containing the unexpected keys.

The given incompatible_keys can be modified inplace if needed.

Note that the checks performed when calling *load_state_dict()* with `strict=True` are affected by modifications the hook makes to `missing_keys` or `unexpected_keys`, as expected. Additions to either set of keys will result in an error being thrown when `strict=True`, and clearning out both missing and unexpected keys will avoid an error.

Returns a handle that can be used to remove the added hook by calling `handle.remove()`

Return type `torch.utils.hooks.RemovableHandle`

register_module(*name: str*, *module: Optional[torch.nn.modules.module.Module]*) → None
Alias for *add_module()*.

register_parameter(*name: str*, *param: Optional[torch.nn.parameter.Parameter]*) → None
Adds a parameter to the module.

The parameter can be accessed as an attribute using given name.

Parameters

- **name** (*str*) – name of the parameter. The parameter can be accessed from this module using the given name
- **param** (*Parameter or None*) – parameter to be added to the module. If `None`, then operations that run on parameters, such as *cuda*, are ignored. If `None`, the parameter is **not** included in the module's *state_dict*.

requires_grad_(*requires_grad: bool = True*) → torch.nn.modules.module.T
Change if autograd should record operations on parameters in this module.

This method sets the parameters' `requires_grad` attributes in-place.

This method is helpful for freezing part of the module for finetuning or training parts of a model individually (e.g., GAN training).

See locally-disable-grad-doc for a comparison between *.requires_grad_()* and several similar mechanisms that may be confused with it.

Parameters **requires_grad** (*bool*) – whether autograd should record operations on parameters in this module. Default: `True`.

Returns self

Return type Module

set_extra_state(*state: Any*)
This function is called from *load_state_dict()* to handle any extra state found within the *state_dict*. Implement this function and a corresponding *get_extra_state()* for your module if you need to store extra state within its *state_dict*.

Parameters **state** (*dict*) – Extra state from the *state_dict*

share_memory() → torch.nn.modules.module.T
See `torch.Tensor.share_memory_()`

state_dict(**args*, *destination=None*, *prefix=''*, *keep_vars=False*)
Returns a dictionary containing references to the whole state of the module.

Both parameters and persistent buffers (e.g. running averages) are included. Keys are corresponding parameter and buffer names. Parameters and buffers set to `None` are not included.

Note: The returned object is a shallow copy. It contains references to the module's parameters and buffers.

Warning: Currently `state_dict()` also accepts positional arguments for `destination`, `prefix` and `keep_vars` in order. However, this is being deprecated and keyword arguments will be enforced in future releases.

Warning: Please avoid the use of argument `destination` as it is not designed for end-users.

Parameters

- **destination** (*dict, optional*) – If provided, the state of module will be updated into the dict and the same object is returned. Otherwise, an `OrderedDict` will be created and returned. Default: `None`.
- **prefix** (*str, optional*) – a prefix added to parameter and buffer names to compose the keys in state_dict. Default: `''`.
- **keep_vars** (*bool, optional*) – by default the `Tensor` s returned in the state dict are detached from autograd. If it's set to `True`, detaching will not be performed. Default: `False`.

Returns a dictionary containing a whole state of the module

Return type dict

Example:

```
>>> # xdoctest: +SKIP("undefined vars")
>>> module.state_dict().keys()
['bias', 'weight']
```

to(**args*, ***kwargs*)

Moves and/or casts the parameters and buffers.

This can be called as

to(*device=None*, *dtype=None*, *non_blocking=False*)

to(*dtype*, *non_blocking=False*)

to(*tensor*, *non_blocking=False*)

to(*memory_format=torch.channels_last*)

Its signature is similar to `torch.Tensor.to()`, but only accepts floating point or complex `dtype`s. In addition, this method will only cast the floating point or complex parameters and buffers to `dtype` (if given). The integral parameters and buffers will be moved `device`, if that is given, but with dtypes unchanged. When `non_blocking` is set, it tries to convert/move asynchronously with respect to the host if possible, e.g., moving CPU Tensors with pinned memory to CUDA devices.

See below for examples.

Note: This method modifies the module in-place.

Parameters

- **device** (`torch.device`) – the desired device of the parameters and buffers in this module
- **dtype** (`torch.dtype`) – the desired floating point or complex dtype of the parameters and buffers in this module
- **tensor** (*torch.Tensor*) – Tensor whose dtype and device are the desired dtype and device for all parameters and buffers in this module

- **memory_format** (torch.memory_format) – the desired memory format for 4D parameters and buffers in this module (keyword only argument)

Returns self

Return type Module

Examples:

```
>>> # xdoctest: +IGNORE_WANT("non-deterministic")
>>> linear = nn.Linear(2, 2)
>>> linear.weight
Parameter containing:
tensor([[ 0.1913, -0.3420],
        [-0.5113, -0.2325]])
>>> linear.to(torch.double)
Linear(in_features=2, out_features=2, bias=True)
>>> linear.weight
Parameter containing:
tensor([[ 0.1913, -0.3420],
        [-0.5113, -0.2325]], dtype=torch.float64)
>>> # xdoctest: +REQUIRES(env:TORCH_DOCTEST_CUDA1)
>>> gpu1 = torch.device("cuda:1")
>>> linear.to(gpu1, dtype=torch.half, non_blocking=True)
Linear(in_features=2, out_features=2, bias=True)
>>> linear.weight
Parameter containing:
tensor([[ 0.1914, -0.3420],
        [-0.5112, -0.2324]], dtype=torch.float16, device='cuda:1')
>>> cpu = torch.device("cpu")
>>> linear.to(cpu)
Linear(in_features=2, out_features=2, bias=True)
>>> linear.weight
Parameter containing:
tensor([[ 0.1914, -0.3420],
        [-0.5112, -0.2324]], dtype=torch.float16)

>>> linear = nn.Linear(2, 2, bias=None).to(torch.cdouble)
>>> linear.weight
Parameter containing:
tensor([[ 0.3741+0.j,  0.2382+0.j],
        [ 0.5593+0.j, -0.4443+0.j]], dtype=torch.complex128)
>>> linear(torch.ones(3, 2, dtype=torch.cdouble))
tensor([[0.6122+0.j, 0.1150+0.j],
        [0.6122+0.j, 0.1150+0.j],
        [0.6122+0.j, 0.1150+0.j]], dtype=torch.complex128)
```

to_empty(*, *device: Union[str, torch.device]*) → torch.nn.modules.module.T

Moves the parameters and buffers to the specified device without copying storage.

Parameters **device** (torch.device) – The desired device of the parameters and buffers in this module.

Returns self

Return type Module

train(*mode: bool = True*) → torch.nn.modules.module.T
Sets the module in training mode.

This has any effect only on certain modules. See documentations of particular modules for details of their behaviors in training/evaluation mode, if they are affected, e.g. `Dropout`, `BatchNorm`, etc.

Parameters **mode** (*bool*) – whether to set training mode (`True`) or evaluation mode (`False`). Default: `True`.

Returns self

Return type Module

training: bool

type(*dst_type: Union[torch.dtype, str]*) → torch.nn.modules.module.T
Casts all parameters and buffers to `dst_type`.

Note: This method modifies the module in-place.

Parameters **dst_type** (*type or string*) – the desired type

Returns self

Return type Module

xpu(*device: Optional[Union[int, torch.device]] = None*) → torch.nn.modules.module.T
Moves all model parameters and buffers to the XPU.

This also makes associated parameters and buffers different objects. So it should be called before constructing optimizer if the module will live on XPU while being optimized.

Note: This method modifies the module in-place.

Parameters **device** (*int, optional*) – if specified, all parameters will be copied to that device

Returns self

Return type Module

zero_grad(*set_to_none: bool = False*) → None
Sets gradients of all model parameters to zero. See similar function under `torch.optim.Optimizer` for more context.

Parameters **set_to_none** (*bool*) – instead of setting to zero, set the grads to None. See `torch.optim.Optimizer.zero_grad()` for details.

display_imgs(*imgs: List[numpy.ndarray]*, *title: Optional[str] = None*) → None

weights_init(*m: torch.nn.modules.module.Module*) → None

PYTHON MODULE INDEX

S

INDEX

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

R

S

T

U

V

W

X

Z